
Lua

目錄

介紹	0
目录	1
索引	2
1 – 简介	3
2 – 基本概念	4
3 – 语言定义	5
4 – 编程接口	6
5 – 辅助库	7
6 – 标准库	8
7 – 独立版 Lua	9
8 – 与之前版本不兼容的地方	10
9 – Lua 的完整语法	11
Lua 5.3 术语中英对照表	12
关于 Lua	13
安装 Lua	14
自 Lua 5.2 以来的变更	15
License	16

Lua 5.3 参考手册



作者 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

译者 云风

最后修改时间: 2015年1月14日23:07

目录

- 1 – 简介
- 2 – 基本概念
  - 2.1 – 值与类型
  - 2.2 – 环境与全局环境
  - 2.3 – 错误处理
  - 2.4 – 元表及元方法
  - 2.5 – 垃圾收集
    - 2.5.1 – 垃圾收集的元方法
    - 2.5.2 – 弱表
  - 2.6 – 协程
- 3 – 语言定义
  - 3.1 – 词法约定
  - 3.2 – 变量
  - 3.3 – 语句
    - 3.3.1 – 语句块
    - 3.3.2 – 代码块
    - 3.3.3 – 赋值
    - 3.3.4 – 控制结构
    - 3.3.5 – For 语句
    - 3.3.6 – 函数调用语句
    - 3.3.7 – 局部声明
  - 3.4 – 表达式
    - 3.4.1 – 数学运算操作符
    - 3.4.2 – 位操作符
    - 3.4.3 – 强制转换
    - 3.4.4 – 比较操作符
    - 3.4.5 – 逻辑操作符
    - 3.4.6 – 字符串连接
    - 3.4.7 – 取长度操作符
    - 3.4.8 – 优先级
    - 3.4.9 – 表构建
    - 3.4.10 – 函数调用
    - 3.4.11 – 函数定义
  - 3.5 – 可见性规则
- 4 – 编程接口
  - 4.1 – 栈

  - 4.2 – 栈大小
  - 4.3 – 有效索引与可接受索引
  - 4.4 – C 闭包
  - 4.5 – 注册表
  - 4.6 – C 中的错误处理
  - 4.7 – C 中的让出处理
  - 4.8 – 函数和类型
  - 4.9 – 调试接口
- 5 – 辅助库
  - 5.1 – 函数和类型
- 6 – 标准库
  - 6.1 – 基本函数
  - 6.2 – 协程管理
  - 6.3 – 模块
  - 6.4 – 字符串处理
    - 6.4.1 – 匹配模式
    - 6.4.2 – 打包和解包用到的格式串
  - 6.5 – UTF-8 支持
  - 6.6 – 表处理
  - 6.7 – 数学函数
  - 6.8 – 输入输出设施
  - 6.9 – 操作系统设施
  - 6.10 – 调试库
- 7 – 独立版 Lua
- 8 – 与之前版本不兼容的地方
  - 8.1 – 语言的变更
  - 8.2 – 库的变更
  - 8.3 – API 的变更
- 9 – Lua 的完整语法

索引

Lua 函数

基础

- _G
- _VERSION
- assert
- collectgarbage
- dofile
- error
- getmetatable
- ipairs
- load
- loadfile
- next
- pairs
- pcall
- print
- rawequal
- rawget
- rawlen
- rawset
- require
- select
- setmetatable
- tonumber
- tostring
- type
- xpcall

协程

- coroutine.create
- coroutine.isyieldable
- coroutine.resume
- coroutine.running

- coroutine.status
- coroutine.wrap
- coroutine.yield

调试

- debug.debug
- debug.gethook
- debug.getinfo
- debug.getlocal
- debug.getmetatable
- debug.getregistry
- debug.getupvalue
- debug.getuservalue
- debug.sethook
- debug.setlocal
- debug.setmetatable
- debug.setupvalue
- debug.setuservalue
- debug.traceback
- debug.upvalueid
- debug.upvaluejoin

输入输出

- io.close
- io.flush
- io.input
- io.lines
- io.open
- io.output
- io.popen
- io.read
- io.stderr
- io.stdin
- io.stdout
- io.tmpfile
- io.type
- io.write
- file:close

- file:flush
- file:lines
- file:read
- file:seek
- file:setvbuf
- file:write

数学

- math.abs
- math.acos
- math.asin
- math.atan
- math.ceil
- math.cos
- math.deg
- math.exp
- math.floor
- math.fmod
- math.huge
- math.log
- math.max
- math.maxinteger
- math.min
- math.mininteger
- math.modf
- math.pi
- math.rad
- math.random
- math.randomseed
- math.sin
- math.sqrt
- math.tan
- math.tointeger
- math.type
- math.ult

操作系统

- os.clock

- os.date
- os.difftime
- os.execute
- os.exit
- os.getenv
- os.remove
- os.rename
- os.setlocale
- os.time
- os.tmpname

包

- package.config
- package.cpath
- package.loaded
- package.loadlib
- package.path
- package.preload
- package.searchers
- package.searchpath

字符串

- string.byte
- string.char
- string.dump
- string.find
- string.format
- string.gmatch
- string.gsub
- string.len
- string.lower
- string.match
- string.pack
- string.packsize
- string.rep
- string.reverse
- string.sub
- string.unpack

- string.upper

表

- table.concat
- table.insert
- table.move
- table.pack
- table.remove
- table.sort
- table.unpack

utf8

- utf8.char
- utf8.charpattern
- utf8.codepoint
- utf8.codes
- utf8.len
- utf8.offset

环境变量

- LUA_CPATH
- LUA_CPATH_5_3
- LUA_INIT
- LUA_INIT_5_3
- LUA_PATH
- LUA_PATH_5_3

C API

- lua_Alloc
- lua_CFunction
- lua_Debug
- lua_Hook
- lua_Integer
- lua_KContext
- lua_KFunction

- lua_Number
- lua_Reader
- lua_State
- lua_Unsigned
- lua_Writer

- lua_absindex

- lua_arith
- lua_atpanic
- lua_call
- lua_callk
- lua_checkstack
- lua_close
- lua_compare
- lua_concat
- lua_copy
- lua_createtable
- lua_dump
- lua_error
- lua_gc
- lua_getallocf
- lua_getextraspace
- lua_getfield
- lua_getglobal
- lua_gethook
- lua_gethookcount
- lua_gethookmask
- lua_geti
- lua_getinfo
- lua_getlocal
- lua_getmetatable
- lua_getstack
- lua_gettable
- lua_gettop
- lua_getupvalue
- lua_getuservalue
- lua_insert
- lua_isboolean
- lua_iscfunction
- lua_isfunction

- lua_isinteger
- lua_islightuserdata
- lua_isnil
- lua_isnone
- lua_isnoneornil
- lua_isnumber
- lua_isstring
- lua_istable
- lua_isthread
- lua_isuserdata
- lua_isyieldable
- lua_len
- lua_load
- lua_newstate
- lua_newtable
- lua_newthread
- lua_newuserdata
- lua_next
- lua_numbertointeger
- lua_pcall
- lua_pcallk
- lua_pop
- lua_pushboolean
- lua_pushcclosure
- lua_pushcfunction
- lua_pushfstring
- lua_pushglobaltable
- lua_pushinteger
- lua_pushlightuserdata
- lua_pushliteral
- lua_pushlstring
- lua_pushnil
- lua_pushnumber
- lua_pushstring
- lua_pushthread
- lua_pushvalue
- lua_pushvfstring
- lua_rawequal
- lua_rawget
- lua_rawgeti

- lua_rawgetp
- lua_rawlen
- lua_rawset
- lua_rawseti
- lua_rawsetp
- lua_register
- lua_remove
- lua_replace
- lua_resume
- lua_rotate
- lua_setallocf
- lua_setfield
- lua_setglobal
- lua_sethook
- lua_seti
- lua_setlocal
- lua_setmetatable
- lua_settable
- lua_settop
- lua_setupvalue
- lua_setuservalue
- lua_status
- lua_stringtonumber
- lua_toboolean
- lua_tocfunction
- lua_tointeger
- lua_tointegerx
- lua_tolstring
- lua_tonumber
- lua_tonumberx
- lua_topointer
- lua_tostring
- lua_tothread
- lua_touserdata
- lua_type
- lua_typename
- lua_upvalueid
- lua_upvalueindex
- lua_upvaluejoin
- lua_version

- lua_xmove
- lua_yield
- lua_yieldk

辅助库

- luaL_Buffer
- luaL_Reg
- luaL_Stream

- luaL_addchar

- luaL_addlstring
- luaL_addsize
- luaL_addstring
- luaL_addvalue
- luaL_argcheck
- luaL_argerror
- luaL_buffinit
- luaL_buffinitsize
- luaL_callmeta
- luaL_checkany
- luaL_checkinteger
- luaL_checklstring
- luaL_checknumber
- luaL_checkoption
- luaL_checkstack
- luaL_checkstring
- luaL_checktype
- luaL_checkudata
- luaL_checkversion
- luaL_dofile
- luaL_dostring
- luaL_error
- luaL_execresult
- luaL_fileresult
- luaL_getmetafield
- luaL_getmetatable
- luaL_getsubtable
- luaL_gsub
- luaL_len

- luaL_loadbuffer
- luaL_loadbufferx
- luaL_loadfile
- luaL_loadfilex
- luaL_loadstring
- luaL_newlib
- luaL_newlibtable
- luaL_newmetatable
- luaL_newstate
- luaL_openlibs
- luaL_optinteger
- luaL_optlstring
- luaL_optnumber
- luaL_optstring
- luaL_prepbuffer
- luaL_prepbuffsize
- luaL_pushresult
- luaL_pushresultsize
- luaL_ref
- luaL_requiref
- luaL_setfuncs
- luaL_setmetatable
- luaL_testudata
- luaL_tolstring
- luaL_traceback
- luaL_typename
- luaL_unref
- luaL_where

标准库

- luaopen_base
- luaopen_coroutine
- luaopen_debug
- luaopen_io
- luaopen_math
- luaopen_os
- luaopen_package
- luaopen_string
- luaopen_table

- luaopen_utf8

常量

- LUA_ERRERR
- LUA_ERRFILE
- LUA_ERRGCMM
- LUA_ERRMEM
- LUA_ERRRUN
- LUA_ERRSYNTAX
- LUA_HOOKCALL
- LUA_HOOKCOUNT
- LUA_HOOKLINE
- LUA_HOOKRET
- LUA_HOOKTAILCALL
- LUA_MASKCALL
- LUA_MASKCOUNT
- LUA_MASKLINE
- LUA_MASKRET
- LUA_MAXINTEGER
- LUA_MININTEGER
- LUA_MINSTACK
- LUA_MULTRET
- LUA_NOREF
- LUA_OK
- LUA_OPADD
- LUA_OPBAND
- LUA_OPBNOT
- LUA_OPBOR
- LUA_OPBXOR
- LUA_OPDIV
- LUA_OPEQ
- LUA_OPIDIV
- LUA_OPLE
- LUA_OPLT
- LUA_OPMOD
- LUA_OPMUL
- LUA_OPPOW
- LUA_OPSHL
- LUA_OPSHR

- LUA_OPSUB
- LUA_OPUNM
- LUA_REFNIL
- LUA_REGISTRYINDEX
- LUA_RIDX_GLOBALS
- LUA_RIDX_MAINTHREAD
- LUA_TBOOLEAN
- LUA_TFUNCTION
- LUA_TLIGHTUSERDATA
- LUA_TNIL
- LUA_TNONE
- LUA_TNUMBER
- LUA_TSTRING
- LUA_TTABLE
- LUA_TTHREAD
- LUA_TUSERDATA
- LUA_USE_APICHECK
- LUA_YIELD
- LUAL_BUFFERSIZE

1 – 简介

Lua 是一门扩展式程序设计语言，被设计成支持通用过程式编程，并有相关数据描述设施。同时对面向对象编程、函数式编程和数据驱动式编程也提供了良好的支持。 它作为一个强大、轻量的嵌入式脚本语言，可供任何需要的程序使用。 Lua 由 *clean C*（标准 C 和 C++ 间共通的子集） 实现成一个库。

作为一门扩展式语言，Lua 没有 "main" 程序的概念： 它只能 嵌入 一个宿主程序中工作， 该宿主程序被称为 被嵌入程序 或者简称 宿主 。 宿主程序可以调用函数执行一小段 Lua 代码，可以读写 Lua 变量，可以注册 C 函数让 Lua 代码调用。 依靠 C 函数，Lua 可以共享相同的语法框架来定制编程语言，从而适用不同的领域。 Lua 的官方发布版包含一个叫做 `lua` 的宿主程序示例， 它是一个利用 Lua 库实现的完整独立的 Lua 解释器，可用于交互式应用或批处理。

Lua 是一个自由软件，其使用许可证决定了它的使用过程无需任何担保。 本手册所描述的实现可以在 Lua 的官方网站 `www.lua.org` 找到。

与其它的许多参考手册一样，这份文档有些地方比较枯燥。 关于 Lua 背后的设计思想， 可以看看 Lua 网站上提供的技术论文。 至于用 Lua 编程的细节介绍， 请参阅 Roberto 的书，*Programming in Lua*。

2 – 基本概念

本章描述了语言的基本概念。

2.1 – 值与类型

Lua 是一门动态类型语言。 这意味着变量没有类型；只有值才有类型。 语言中不设类型定义。 所有的值携带自己的类型。

Lua 中所有的值都是 一等公民。 这意味着所有的值均可保存在变量中、 当作参数传递给其它函数、以及作为返回值。

Lua 中有八种基本类型： *nil*、*boolean*、*number*、*string*、*function*、*userdata*、 *thread* 和 *table*。 *Nil* 是值 **nil** 的类型， 其主要特征就是和其它值区别开；通常用来表示一个有意义的值不存在时的状态。 *Boolean* 是 **false** 与 **true** 两个值的类型。 **nil** 和 **false** 都会导致条件判断为假； 而其它任何值都表示为真。 *Number* 代表了整数和实数（浮点数）。 *String* 表示一个不可变的字节序列。 Lua 对 8 位是友好的： 字符串可以容纳任意 8 位值， 其中包含零 ('`\0`') 。 Lua 的字符串与编码无关； 它不关心字符串中具体内容。

number 类型有两种内部表现方式， 整数 和 浮点数。 对于何时使用哪种内部形式，Lua 有明确的规则， 但它也按需（参见 §3.4.3）作自动转换。 因此，程序员多数情况下可以选择忽略整数与浮点数之间的差异或者假设完全控制每个数字的内部表现方式。 标准 Lua 使用 64 位整数和双精度（64 位）浮点数， 但你也可以把 Lua 编译成使用 32 位整数和单精度（32 位）浮点数。 以 32 位表示数字对小型机器以及嵌入式系统特别合适。 （参见 `luaconf.h` 文件中的宏 `LUA_32BITS` 。）

Lua 可以调用（以及操作）用 Lua 或 C （参见 §3.4.10）编写的函数。 这两种函数有统一类型 *function*。

userdata 类型允许将 C 中的数据保存在 Lua 变量中。 用户数据类型的值是一个内存块， 有两种用户数据： 完全用户数据 ，指一块由 Lua 管理的内存对应的对象； 轻量用户数据 ，则指一个简单的 C 指针。 用户数据在 Lua 中除了赋值与相等性判断之外没有其他预定义的操作。 通过使用 元表 ，程序员可以给完全用户数据定义一系列的操作 （参见 §2.4）。 你只能通过 C API 而无法在 Lua 代码中创建或者修改用户数据的值， 这保证了数据仅被宿主程序所控制。

thread 类型表示了一个独立的执行序列，被用于实现协程 （参见 §2.6）。 Lua 的线程与操作系统的线程毫无关系。 Lua 为所有的系统，包括那些不支持原生线程的系统，提供了协程支持。

table 是一个关联数组， 也就是说，这个数组不仅仅以数字做索引，除了 **nil** 和 NaN 之外的所有 Lua 值 都可以做索引。 （*Not a Number* 是一个特殊的数字，它用于表示未定义或表示不了的运算结果，比如 `0/0` 。） 表可以是 异构 的； 也就是说，表内可以包含任何类型的值（ **nil** 除外）。 任何键的值若为 **nil** 就不会被记入表结构内部。 换言之， 对于表内不存在的键，都对应着值 **nil** 。

表是 Lua 中唯一的数据结构， 它可被用于表示普通数组、序列、符号表、集合、记录、图、树等等。 对于记录，Lua 使用域名作为索引。 语言提供了 `a.name` 这样的语法糖来替代 `a["name"]` 这种写法以方便记录这种结构的使用。 在 Lua 中有多种便利的方式创建表（参见 §3.4.9）。

我们使用 序列 这个术语来表示一个用 {1..*n*} 的正整数集做索引的表。 这里的非负整数 *n* 被称为该序列的长度（参见 §3.4.7）。

和索引一样，表中每个域的值也可以是任何类型。 需要特别指出的是：既然函数是一等公民，那么表的域也可以是函数。 这样，表就可以携带 方法 了。 （参见 §3.4.11）。

索引一张表的原则遵循语言中的直接比较规则。 当且仅当 `i` 与 `j` 直接比较相等时 （即不通过元方法的比较）， 表达式 `a[i]` 与 `a[j]` 表示了表中相同的元素。 特别指出：一个可以完全表示为整数的浮点数和对应的整数相等 （例如：`1.0 == 1`）。 为了消除歧义，当一个可以完全表示为整数的浮点数做为键值时， 都会被转换为对应的整数储存。 例如，当你写 `a[2.0] = true` 时， 实际被插入表中的键是整数 `2` 。 （另一方面，2 与 "`2`" 是两个不同的 Lua 值， 故而它们可以是同一张表中的不同项。）

表、函数、线程、以及完全用户数据在 Lua 中被称为 对象： 变量并不真的 持有 它们的值，而仅保存了对这些对象的 引用。 赋值、参数传递、函数返回，都是针对引用而不是针对值的操作， 这些操作均不会做任何形式的隐式拷贝。

库函数 `type` 用于以字符串形式返回给定值的类型。 （参见 §6.1）。

2.2 – 环境与全局环境

后面在 §3.2 以及 §3.3.3 会讨论， 引用一个叫 `var` 的自由名字（指在任何层级都未被声明的名字） 在句法上都被翻译为 `_ENV.var` 。 此外，每个被编译的 Lua 代码块都会有一个外部的局部变量叫 `_ENV` （参见 §3.3.2）， 因此， `_ENV` 这个名字永远都不会成为一个代码块中的自由名字。

在转译那些自由名字时， `_ENV` 是否是那个外部的局部变量无所谓。 `_ENV` 和其它你可以使用的变量名没有区别。 这里特别指出，你可以定义一个新变量或指定一个参数叫这个名字。 当编译器在转译自由名字时所用到的 `_ENV` ， 指的是你的程序在那个点上可见的那个名为 _ENV 的变量。 （Lua 的可见性规则参见 §3.5）

被 `_ENV` 用于值的那张表被称为 环境。

Lua 保有一个被称为 全局环境 特别环境。它被保存在 C 注册表 （参见 §4.5）的一个特别索引下。 在 Lua 中，全局变量 `_G` 被初始化为这个值。 （ `_G` 不被内部任何地方使用。）

当 Lua 加载一个代码块，`_ENV` 这个上值的默认值就是这个全局环境 （参见 `load` ）。 因此，在默认情况下，Lua 代码中提及的自由名字都指的全局环境中的相关项 （因此，它们也被称为 全局变量 ）。 此外，所有的标准库都被加载入全局环境，一些函数也针对这个环境做操作。 你可以用 `load` （或 `loadfile` ）加载代码块，并赋予它们不同的环境。 （在 C 里，当你加载一个代码块后，可以通过改变它的第一个上值来改变它的环境。）

2.3 – 错误处理

由于 Lua 是一门嵌入式扩展语言，其所有行为均源于宿主程序中 C 代码对某个 Lua 库函数的调用。 （单独使用 Lua 时， `lua` 程序就是宿主程序。） 所以，在编译或运行 Lua 代码块的过程中，无论何时发生错误， 控制权都返回给宿主，由宿主负责采取恰当的措施（比如打印错误消息）。

可以在 Lua 代码中调用 `error` 函数来显式地抛出一个错误。 如果你需要在 Lua 中捕获这些错误， 可以使用 `pcall` 或 `xpcall` 在 保护模式 下调用一个函数。

无论何时出现错误，都会抛出一个携带错误信息的 错误对象 （错误消息）。 Lua 本身只会为错误生成字符串类型的错误对象， 但你的程序可以为错误生成任何类型的错误对象， 这就看你的 Lua 程序或宿主程序如何处理这些错误对象。

使用 `xpcall` 或 `lua_pcall` 时， 你应该提供一个 消息处理函数 用于错误抛出时调用。 该函数需接收原始的错误消息，并返回一个新的错误消息。 它在错误发生后栈尚未展开时调用， 因此可以利用栈来收集更多的信息， 比如通过探知栈来创建一组栈回溯信息。 同时，该处理函数也处于保护模式下，所以该函数内发生的错误会再次触发它（递归）。 如果递归太深， Lua 会终止调用并返回一个合适的消息。

2.4 – 元表及元方法

Lua 中的每个值都可以有一个 元表。 这个 元表 就是一个普通的 Lua 表， 它用于定义原始值在特定操作下的行为。 如果你想改变一个值在特定操作下的行为，你可以在它的元表中设置对应域。 例如，当你对非数字值做加操作时， Lua 会检查该值的元表中的 "`__add`" 域下的函数。 如果能找到，Lua 则调用这个函数来完成加这个操作。

元表中的键对应着不同的 事件 名； 键关联的那些值被称为 元方法。 在上面那个例子中引用的事件为 `"add"` ， 完成加操作的那个函数就是元方法。

你可以用 `getmetatable` 函数 来获取任何值的元表。

使用 `setmetatable` 来替换一张表的元表。在 Lua 中，你不可以改变表以外其它类型的值的元表（除非你使用调试库（参见§6.10））；若想改变这些非表类型的值的元表，请使用 C API。

表和完全用户数据有独立的元表（当然，多个表和用户数据可以共享同一个元表）。其它类型的值按类型共享元表；也就是说所有的数字都共享同一个元表，所有的字符串共享另一个元表等等。默认情况下，值是没有元表的，但字符串库在初始化的时候为字符串类型设置了元表（参见 §6.4）。

元表决定了一个对象在数学运算、位运算、比较、连接、取长度、调用、索引时的行为。元表还可以定义一个函数，当表对象或用户数据对象在垃圾回收（参见§2.5）时调用它。

接下来会给出一张元表可以控制的事件的完整列表。每个操作都用对应的事件名来区分。每个事件的键名用加有 '`__`' 前缀的字符串来表示；例如 "add" 操作的键名为字符串 "`__add`"。注意、Lua 从元表中直接获取元方法；访问元表中的元方法永远不会触发另一次元方法。下面的代码模拟了 Lua 从一个对象 `obj` 中获取一个元方法的过程：

```
rawget(getmetatable(obj) or {}, "__" .. event_name)
```

对于一元操作符（取负、求长度、位反），元方法调用的时候，第二个参数是个哑元，其值等于第一个参数。这样处理仅仅是为了简化 Lua 的内部实现（这样处理可以让所有的操作都和二元操作一致），这个行为有可能在将来的版本中移除。（使用这个额外参数的行为都是不确定的。）

- **"add":** `+` 操作。如果任何不是数字的值（包括不能转换为数字的字符串）做加法，Lua 就会尝试调用元方法。首先、Lua 检查第一个操作数（即使它是合法的），如果这个操作数没有为 "`__add`" 事件定义元方法，Lua 就会接着检查第二个操作数。一旦 Lua 找到了元方法，它将把两个操作数作为参数传入元方法，元方法的结果（调整为单个值）作为这个操作的结果。如果找不到元方法，将抛出一个错误。
- **"sub":** `-` 操作。行为和 "add" 操作类似。
- **"mul":** `*` 操作。行为和 "add" 操作类似。
- **"div":** `/` 操作。行为和 "add" 操作类似。
- **"mod":** `%` 操作。行为和 "add" 操作类似。
- **"pow":** `^` （次方）操作。行为和 "add" 操作类似。
- **"unm":** `-` （取负）操作。行为和 "add" 操作类似。
- **"idiv":** `//` （向下取整除法）操作。行为和 "add" 操作类似。
- **"band":** `&` （按位与）操作。行为和 "add" 操作类似，不同的是 Lua 会在任何一个操作数无法转换为整数时（参见 §3.4.3）尝试取元方法。
- **"bor":** `|` （按位或）操作。行为和 "band" 操作类似。
- **"bxor":** `~` （按位异或）操作。行为和 "band" 操作类似。
- **"bnot":** `~` （按位非）操作。行为和 "band" 操作类似。
- **"shl":** `<<` （左移）操作。行为和 "band" 操作类似。

- **"shr":** `>>` （右移）操作。 行为和 "band" 操作类似。
- **"concat":** `..` （连接）操作。 行为和 "add" 操作类似， 不同的是 Lua 在任何操作数即不是一个字符串 也不是数字（数字总能转换为对应的字符串）的情况下尝试元方法。
- **"len":** `#` （取长度）操作。 如果对象不是字符串，Lua 会尝试它的元方法。 如果有元方法，则调用它并将对象以参数形式传入， 而返回值（被调整为单个）则作为结果。 如果对象是一张表且没有元方法， Lua 使用表的取长度操作（参见 §3.4.7）。 其它情况，均抛出错误。
- **"eq":** `==` （等于）操作。 和 "add" 操作行为类似， 不同的是 Lua 仅在两个值都是表或都是完全用户数据 且它们不是同一个对象时才尝试元方法。 调用的结果总会被转换为布尔量。
- **"lt":** `<` （小于）操作。 和 "add" 操作行为类似， 不同的是 Lua 仅在两个值不全为整数也不全为字符串时才尝试元方法。 调用的结果总会被转换为布尔量。
- **"le":** `<=` （小于等于）操作。 和其它操作不同， 小于等于操作可能用到两个不同的事件。 首先，像 "lt" 操作的行为那样，Lua 在两个操作数中查找 " `__le` " 元方法。 如果一个元方法都找不到，就会再次查找 " `__lt` " 事件， 它会假设 `a <= b` 等价于 `not (b < a)` 。 而其它比较操作符类似，其结果会被转换为布尔量。
- **"index":** 索引 `table[key]` 。 当 `table` 不是表或是表 `table` 中不存在 `key` 这个键时，这个事件被触发。 此时，会读出 `table` 相应的元方法。

  尽管名字取成这样， 这个事件的元方法其实可以是一个函数也可以是一张表。 如果它是一个函数，则以 `table` 和 `key` 作为参数调用它。 如果它是一张表，最终的结果就是以 `key` 取索引这张表的结果。 （这个索引过程是走常规的流程，而不是直接索引， 所以这次索引有可能引发另一次元方法。）

- **"newindex":** 索引赋值 `table[key] = value` 。 和索引事件类似，它发生在 `table` 不是表或是表 `table` 中不存在 `key` 这个键的时候。 此时，会读出 `table` 相应的元方法。

  同索引过程那样， 这个事件的元方法即可以是函数，也可以是一张表。 如果是一个函数， 则以 `table` 、 `key` 、以及 `value` 为参数传入。 如果是一张表， Lua 对这张表做索引赋值操作。 （这个索引过程是走常规的流程，而不是直接索引赋值， 所以这次索引赋值有可能引发另一次元方法。）

  一旦有了 "newindex" 元方法， Lua 就不再做最初的赋值操作。 （如果有必要，在元方法内部可以调用 `rawset` 来做赋值。）

- **"call":** 函数调用操作 `func(args)` 。 当 Lua 尝试调用一个非函数的值的时候会触发这个事件 （即 `func` 不是一个函数）。 查找 `func` 的元方法， 如果找得到，就调用这个元方法， `func` 作为第一个参数传入，原来调用的参数（ `args` ）后依次排在后面。

2.5 – 垃圾收集

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来， 也不用考虑在对象不再被使用后怎样释放它们所占用的内存。 Lua 运行了一个 垃圾收集器 来收集所有 死对象 （即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。 Lua 中所有用到的内存，如：字符串、表、用户数据、函数、线程、 内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。 它使用这两个数字来控制垃圾收集循环： 垃圾收集器间歇率 和 垃圾收集器步进倍率。 这两个数字都使用百分数为单位 （例如：值 100 在内部表示 1 ）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。 增大这个值会减少收集器的积极性。 当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。 设置这个值为 200 就会让收集器等到总内存使用量达到 之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。 增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。 不要把这个值设得小于 100 ， 那样的话收集器就工作的太慢了以至于永远都干不完一个循环。 默认值是 200 ， 这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字 （比你的程序可能用到的字节数还大 10% ）， 收集器的行为就像一个 stop-the-world 收集器。 接着你若把间歇率设为 200 ， 收集器的行为就和过去的 Lua 版本一样了： 每次 Lua 使用的内存翻倍时，就做一次完整的收集。

你可以通过在 C 中调用 `lua_gc` 或在 Lua 中调用 `collectgarbage` 来改变这俩数字。 这两个函数也可以用来直接控制收集器（例如停止它或重启它）。

2.5.1 – 垃圾收集元方法

你可以为表设定垃圾收集的元方法， 对于完全用户数据（参见 §2.4）， 则需要使用 C API 。 该元方法被称为 终结器。 终结器允许你配合 Lua 的垃圾收集器做一些额外的资源管理工作 （例如关闭文件、网络或数据库连接，或是释放一些你自己的内存）。

如果要让一个对象（表或用户数据）在收集过程中进入终结流程， 你必须 标记 它需要触发终结器。 当你为一个对象设置元表时，若此刻这张元表中用一个以字符串 "`__gc`" 为索引的域，那么就标记了这个对象需要触发终结器。 注意：如果你给对象设置了一个没有 `__gc` 域的元表，之后才给元表加上这个域， 那么这个对象是没有被标记成需要触发终结器的。 然而，一旦对象被标记， 你还是可以自由的改变其元表中的 `__gc` 域的。

当一个被标记的对象成为了垃圾后， 垃圾收集器并不会立刻回收它。 取而代之的是，Lua 会将其置入一个链表。 在收集完成后，Lua 将遍历这个链表。 Lua 会检查每个链表中的对象的 `__gc` 元方法：如果是一个函数，那么就以对象为唯一参数调用它； 否则直接忽略它。

在每次垃圾收集循环的最后阶段， 本次循环中检测到的需要被回收之对象， 其终结器的触发次序按当初给对象作需要触发终结器的标记之次序的逆序进行； 这就是说，第一个被调用的终结器是程序中最后一个被标记的对象所携的那个。 每个终结器的运行可能发生在执行常规代码过程中的任意一刻。

由于被回收的对象还需要被终结器使用， 该对象（以及仅能通过它访问到的其它对象）一定会被 Lua 复活。 通常，复活是短暂的，对象所属内存会在下一个垃圾收集循环释放。 然后，若终结器又将对象保存去一些全局的地方 （例如：放在一个全局变量里），这次复活就持续生效了。 此外，如果在终结器中对一个正进入终结流程的对象再次做一次标记让它触发终结器， 只要这个对象在下个循环中依旧不可达，它的终结函数还会再调用一次。 无论是哪种情况， 对象所属内存仅在垃圾收集循环中该对象不可达且 没有被标记成需要触发终结器才会被释放。

当你关闭一个状态机（参见 `lua_close`）， Lua 将调用所有被标记了需要触发终结器对象的终结过程， 其次序为标记次序的逆序。 在这个过程中，任何终结器再次标记对象的行为都不会生效。

2.5.2 – 弱表

弱表 指内部元素为 弱引用 的表。 垃圾收集器会忽略掉弱引用。 换句话说，如果一个对象只被弱引用引用到， 垃圾收集器就会回收这个对象。

一张弱表可以有弱键或是弱值，也可以键值都是弱引用。 仅含有弱键的表允许收集器回收它的键，但会阻止对值所指的对象被回收。 若一张表的键值均为弱引用， 那么收集器可以回收其中的任意键和值。 任何情况下，只要键或值的任意一项被回收， 相关联的键值对都会从表中移除。 一张表的元表中的 `__mode` 域控制着这张表的弱属性。 当 `__mode` 域是一个包含字符 '`k`' 的字符串时，这张表的所有键皆为弱引用。 当 `__mode` 域是一个包含字符 '`v`' 的字符串时，这张表的所有值皆为弱引用。

属性为弱键强值的表也被称为 暂时表。 对于一张暂时表， 它的值是否可达仅取决于其对应键是否可达。 特别注意，如果表内的一个键仅仅被其值所关联引用， 这个键值对将被表内移除。

对一张表的弱属性的修改仅在下次收集循环才生效。 尤其是当你把表由弱改强，Lua 还是有可能在修改生效前回收表内一些项目。

只有那些有显式构造过程的对象才会从弱表中移除。 值，例如数字和轻量 C 函数，不受垃圾收集器管辖， 因此不会从弱表中移除 （除非它们的关联项被回收）。 虽然字符串受垃圾回收器管辖， 但它们没有显式的构造过程，所以也不会从弱表中移除。

弱表针对复活的对象 （指那些正在走终结流程，仅能被终结器访问的对象） 有着特殊的行为。 弱值引用的对象，在运行它们的终结器前就被移除了， 而弱键引用的对象则要等到终结器运行完毕后，到下次收集当对象真的被释放时才被移除。 这个行为使得终结器运行时得以访问到由该对象在弱表中所关联的属性。

如果一张弱表在当次收集循环内的复活对象中， 那么在下个循环前这张表有可能未被正确地清理。

2.6 – 协程

Lua 支持协程，也叫 协同式多线程。 一个协程在 Lua 中代表了一段独立的执行线程。 然而，与多线程系统中的线程的区别在于， 协程仅在显式调用一个让出（yield）函数时才挂起当前的执行。

调用函数 `coroutine.create` 可创建一个协程。 其唯一的参数是该协程的主函数。 `create` 函数只负责新建一个协程并返回其句柄 （一个 *thread* 类型的对象）； 而不会启动该协程。

调用 `coroutine.resume` 函数执行一个协程。 第一次调用 `coroutine.resume` 时，第一个参数应传入 `coroutine.create` 返回的线程对象，然后协程从其主函数的第一行开始执行。 传递给 `coroutine.resume` 的其他参数将作为协程主函数的参数传入。 协程启动之后，将一直运行到它终止或 让出。

协程的运行可能被两种方式终止： 正常途径是主函数返回 （显式返回或运行完最后一条指令）； 非正常途径是发生了一个未被捕获的错误。 对于正常结束， `coroutine.resume` 将返回 **true**， 并接上协程主函数的返回值。 当错误发生时， `coroutine.resume` 将返回 **false** 与错误消息。

通过调用 `coroutine.yield` 使协程暂停执行，让出执行权。 协程让出时，对应的最近 `coroutine.resume` 函数会立刻返回，即使该让出操作发生在内嵌函数调用中 （即不在主函数，但在主函数直接或间接调用的函数内部）。 在协程让出的情况下， `coroutine.resume` 也会返回 **true**， 并加上传给 `coroutine.yield` 的参数。 当下次重启同一个协程时， 协程会接着从让出点继续执行。 调用 `coroutine.yield` 会返回任何传给 `coroutine.resume` 的第一个参数之外的其他参数。

与 `coroutine.create` 类似， `coroutine.wrap` 函数也会创建一个协程。 不同之处在于，它不返回协程本身，而是返回一个函数。 调用这个函数将启动该协程。 传递给该函数的任何参数均当作 `coroutine.resume` 的额外参数。 `coroutine.wrap` 返回 `coroutine.resume` 的所有返回值，除了第一个返回值（布尔型的错误码）。 和 `coroutine.resume` 不同， `coroutine.wrap` 不会捕获错误； 而是将任何错误都传播给调用者。

下面的代码展示了一个协程工作的范例：

```
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo(a+1)
      print("co-body", r)
      local r, s = coroutine.yield(a+b, a-b)
      print("co-body", r, s)
      return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

当你运行它，将产生下列输出：

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

你也可以通过 C API 来创建及操作协程：参见函数 `lua_newthread`， `lua_resume`， 以及 `lua_yield`。

3 – 语言定义

这一章描述了 Lua 的词法、语法和句法。 换句话说，本章描述哪些符记是有效的， 它们如何被组合起来，这些组合方式有什么含义。

关于语言的构成概念将用常见的扩展 BNF 表达式写出。 也就是这个样子： {*a*} 表示 0 或多个 *a*， [*a*] 表示一个可选的 *a*。 可以被分解的非最终符号会这样写 non-terminal， 关键字会写成这样 **kword**， 而其它不能被分解的最终符号则写成这样 ‘**=**’ 。 完整的 Lua 语法可以在本手册最后一章 §9 找到。

3.1 – 词法约定

Lua 语言的格式自由。 它会忽略语法元素（符记）间的空格（包括换行）和注释， 仅把它们看作为名字和关键字间的分割符。

Lua 中的 名字 （也被称为 标识符） 可以是由非数字打头的任意字母下划线和数字构成的字符串。 标识符可用于对变量、表的域、以及标签命名。

下列 关键字 是保留的，不可用于名字：

```
     and       break     do        else      elseif    end
     false     for       function  goto      if        in
     local     nil       not       or        repeat    return
     then      true      until     while
```

Lua 语言对大小写敏感： `and` 是一个保留字，但 `And` 与 `AND` 则是两个不同的有效名字。 作为一个约定，程序应避免创建以下划线加一个或多个大写字母构成的名字 （例如 `_VERSION`）。

下列字符串是另外一些符记：

```
     +     -     *     /     %     ^     #
     &     ~     |     <<    >>    //
     ==    ~=    <=    >=    <     >     =
     (     )     {     }     [     ]     ::
     ;     :     ,     .     ..    ...
```

字面串 可以用单引号或双引号括起。 字面串内部可以包含下列 C 风格的转义串： '`\a`' （响铃）， '`\b`' （退格）， '`\f`' （换页）， '`\n`' （换行）， '`\r`' （回车）， '`\t`' （横项制表）， '`\v`' （纵向制表）， '`\\`' （反斜杠）， '`\"`' （双引号）， 以及 '`\'`'(单引号)。 在反斜杠后跟一个真正的换行等价于在字符串中写一个换行符。 转义串 '`\z`' 会忽略其后的一系列空白符，包括换行； 它在你需要对一个很长的字符串常量断行为多行并希望在每个新行保持缩进时非常有用。

Lua 中的字符串可以保存任意 8 位值，其中包括用 '`\0`' 表示的 0 。 一般而言，你可以用字符的数字值来表示这个字符。 方式是用转义串 `\x_XX_`， 此处的 *XX* 必须是恰好两个字符的 16 进制数。 或者你也可以使用转义串 `\_ddd_` ， 这里的 *ddd* 是一到三个十进制数字。 （注意，如果在转义符后接着恰巧是一个数字符号的话， 你就必须在这个转义形式中写满三个数字。）

对于用 UTF-8 编码的 Unicode 字符，你可以用 转义符 `\u{_XXX_}` 来表示 （这里必须有一对花括号）， 此处的 *XXX* 是用 16 进制表示的字符编号。

字面串还可以用一种 长括号 括起来的方式定义。 我们把两个正的方括号间插入 *n* 个等号定义为 第 *n* 级开长括号。 就是说，0 级开的长括号写作 `[[` ， 一级开长括号写作 `[=[` ， 如此等等。 闭长括号也作类似定义； 举个例子，4 级反的长括号写作 `]====]` 。 一个 长字面串 可以由任何一级的开长括号开始，而由第一个碰到的同级的闭长括号结束。 这种方式描述的字符串可以包含任何东西，当然特定级别的反长括号除外。 整个词法分析过程将不受分行限制，不处理任何转义符，并且忽略掉任何不同级别的长括号。 其中碰到的任何形式的换行串（回车、换行、回车加换行、换行加回车），都会被转换为单个换行符。

字面串中的每个不被上述规则影响的字节都呈现为本身。 然而，Lua 是用文本模式打开源文件解析的， 一些系统的文件操作函数对某些控制字符的处理可能有问题。 因此，对于非文本数据，用引号括起来并显式按转义符规则来表述更安全。

为了方便起见， 当一个开长括号后紧接一个换行符时， 这个换行符不会放在字符串内。 举个例子，假设一个系统使用 ASCII 码 （此时 '`a`' 编码为 97 ， 换行编码为 10 ， '`1`' 编码为 49 ）， 下面五种方式描述了完全相同的字符串：

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

数字常量 （或称为 数字量） 可以由可选的小数部分和可选的十为底的指数部分构成， 指数部分用字符 '`e`' 或 '`E`' 来标记。 Lua 也接受以 `0x` 或 `0X` 开头的 16 进制常量。 16 进制常量也接受小数加指数部分的形式，指数部分是以二为底， 用字符 '`p`' 或 '`P`' 来标记。 数字常量中包含小数点或指数部分时，被认为是一个浮点数； 否则被认为是一个整数。 下面有一些合法的整数常量的例子：

```
3   345   0xff   0xBEBADA
```

以下为合法的浮点常量：

```
3.0     3.1416     314.16e-2     0.31416E1     34e1
0x0.1E  0xA23p-4   0X1.921FB54442D18P+1
```

在字符串外的任何地方出现以双横线 (`--`) 开头的部分是 注释 。 如果 `--` 后没有紧跟着一个开大括号， 该注释为 短注释， 注释到当前行末截至。 否则，这是一段 长注释 ， 注释区一直维持到对应的闭长括号。 长注释通常用于临时屏蔽掉一大段代码。

3.2 – 变量

变量是储存值的地方。 Lua 中有三种变量： 全局变量、局部变量和表的域。

单个名字可以指代一个全局变量也可以指代一个局部变量 （或者是一个函数的形参，这是一种特殊形式的局部变量）。

```
var ::= Name
```

名字指 §3.1 中定义的标识符。

所有没有显式声明为局部变量（参见 §3.3.7） 的变量名都被当做全局变量。 局部变量有其 作用范围 ： 局部变量可以被定义在它作用范围中的函数自由使用（参见 §3.5）。

在变量的首次赋值之前， 变量的值均为 **nil**。

方括号被用来对表作索引：

```
var ::= prefixexp '**[**' exp '**]**'
```

对全局变量以及表的域之访问的含义可以通过元表来改变。 以索引方式访问一个变量 `t[i]` 等价于 调用 `gettable_event(t,i)` 。 （参见 §2.4 ，有一份完整的关于 `gettable_event` 函数的说明。 这个函数并没有在 lua 中定义出来，也不能在 lua 中调用。这里我们把提到它只是方便说明问题。）

`var.Name` 这种语法只是一个语法糖，用来表示 `var["Name"]` ：

```
var ::= prefixexp '**.**' Name
```

对全局变量 `x` 的操作等价于操作 `_ENV.x` 。 由于代码块编译的方式， `_ENV` 永远也不可能是一个全局名字 （参见 §2.2）。

3.3 – 语句

Lua 支持所有与 Pascal 或是 C 类似的常见形式的语句， 这个集合包括赋值，控制结构，函数调用，还有变量声明。

3.3.1 – 语句块

语句块是一个语句序列，它们会按次序执行：

```
block ::= {stat}
```

Lua 支持 空语句， 你可以用分号分割语句，也可以以分号开始一个语句块， 或是连着写两个分号：

```
stat ::= ‘**;**’
```

函数调用和赋值语句都可能以一个小括号打头， 这可能让 Lua 的语法产生歧义。 我们来看看下面的代码片断：

```
a = b + c
(print or io.write)('done')
```

从语法上说，可能有两种解释方式：

```
a = b + c(print or io.write)('done')

a = b + c; (print or io.write)('done')
```

当前的解析器总是用第一种结构来解析， 它会将开括号看成函数调用的参数传递开始处。 为了避免这种二义性， 在一条语句以小括号开头时，前面放一个分号是个好习惯：

```
;(print or io.write)('done')
```

一个语句块可以被显式的定界为单条语句：

```
stat ::= **do** block **end**
```

显式的对一个块定界通常用来控制内部变量声明的作用域。 有时，显式定界也用于在一个语句块中间插入 **return** （参见 §3.3.4）。

3.3.2 – 代码块

Lua 的一个编译单元被称为一个 代码块。 从句法构成上讲，一个代码块就是一个语句块。

```
chunk ::= block
```

Lua 把一个代码块当作一个拥有不定参数的匿名函数 （参见§3.4.11）来处理。 正是这样，代码块内可以定义局部变量，它可以接收参数，返回若干值。 此外，这个匿名函数在编译时还为它的作用域绑定了一个外部局部变量 `_ENV` （参见 §2.2）。 该函数总是把 `_ENV` 作为它

唯一的一个上值， 即使这个函数不使用这个变量，它也存在。

代码块可以被保存在文件中，也可以作为宿主程序内部的一个字符串。 要执行一个代码块，首先要让 Lua 加载 它， 将代码块中的代码预编译成虚拟机中的指令， 而后，Lua 用虚拟机解释器来运行编译后的代码。

代码块可以被预编译为二进制形式； 参见程序 `luac` 以及函数 `string.dump` 可获得更多细节。 用源码表示的程序和编译后的形式可自由替换； Lua 会自动检测文件格式做相应的处理 （参见 `load` ）。

3.3.3 – 赋值

Lua 允许多重赋值。 因此，赋值的语法定义是等号左边放一个变量列表， 而等号右边放一个表达式列表。 两边的列表中的元素都用逗号间开：

```
stat ::= varlist ‘**=**’ explist
varlist ::= var {‘**,**’ var}
explist ::= exp {‘**,**’ exp}
```

表达式放在 §3.4 中讨论。

在作赋值操作之前， 那值列表会被 调整 为左边变量列表的个数。 如果值比需要的更多的话，多余的值就被扔掉。 如果值的数量不够需求， 将会按所需扩展若干个 **nil**。 如果表达式列表以一个函数调用结束， 这个函数所返回的所有值都会在调整操作之前被置入值列表中 （除非这个函数调用被用括号括了起来；参见 §3.4）。

赋值语句首先让所有的表达式完成运算， 之后再做赋值操作。 因此，下面这段代码

```
i = 3
i, a[i] = i+1, 20
```

会把 `a[3]` 设置为 20，而不会影响到 `a[4]` 。 这是因为 `a[i]` 中的 `i` 在被赋值为 4 之前就被计算出来了（当时是 3 ）。 简单说 ， 这样一行

```
x, y = y, x
```

会交换 `x` 和 `y` 的值， 及

```
x, y, z = y, z, x
```

会轮换 `x` ， `y` ， `z` 的值。

对全局变量以及表的域的赋值操作的含义可以通过元表来改变。 对 `t[i] = val` 这样的变量索引赋值， 等价于 `settable_event(t,i,val)` 。 （关于函数 `settable_event` 的详细说明，参见 §2.4。 这个函数并没有在 Lua 中定义出来，也不可以被调用。 这里我们列出来，仅仅出于方便解释的目的。）

对于全局变量 `x = val` 的赋值等价于 `_ENV.x = val` （参见 §2.2）。

3.3.4 – 控制结构

if, **while**, and **repeat** 这些控制结构符合通常的意义，而且也有类似的语法：

```
stat ::= **while** exp **do** block **end**
stat ::= **repeat** block **until** exp
stat ::= **if** exp **then** block {**elseif** exp **then** block} [**else** block] *
```

Lua 也有一个 **for** 语句，它有两种形式（参见 §3.3.5）。

控制结构中的条件表达式可以返回任何值。 **false** 与 **nil** 两者都被认为是假。 所有不同于 **nil** 与 **false** 的其它值都被认为是真（特别需要注意的是，数字 0 和空字符串也被认为是真）。

在 **repeat–until** 循环中， 内部语句块的结束点不是在 **until** 这个关键字处， 它还包括了其后的条件表达式。 因此，条件表达式中可以使用循环内部语句块中的定义的局部变量。

goto 语句将程序的控制点转移到一个标签处。 由于句法上的原因， Lua 里的标签也被认为是语句：

```
stat ::= **goto** Name
stat ::= label
label ::= ‘**::**’ Name ‘**::**’
```

除了在内嵌函数中，以及在内嵌语句块中定义了同名标签，的情况外， 标签对于它定义所在的整个语句块可见。 只要 goto 没有进入一个新的局部变量的作用域，它可以跳转到任意可见标签处。

标签和没有内容的语句被称为空语句，它们不做任何操作。

break 被用来结束 **while**、 **repeat**、或 **for** 循环， 它将跳到循环外接着之后的语句运行：

```
stat ::= **break**
```

break 跳出最内层的循环。

return 被用于从函数或是代码块（其实它就是一个函数） 中返回值。 函数可以返回不止一个值，所以 **return** 的语法为

```
stat ::= **return** [explist] [‘**;**’]
```

return 只能被写在一个语句块的最后一句。 如果你真的需要从语句块的中间 **return**， 你可以使用显式的定义一个内部语句块， 一般写作 `do return end` 。 可以这样写是因为现在 **return** 成了（内部）语句块的最后一句了。

3.3.5 – For 语句

for 有两种形式：一种是数字形式， 另一种是通用形式。

数字形式的 **for** 循环，通过一个数学运算不断地运行内部的代码块。 下面是它的语法：

```
stat ::= **for** Name ‘**=**’ exp ‘**,**’ exp [‘**,**’ exp] **do** block **end**
```

block 将把 *name* 作循环变量。 从第一个 *exp* 开始起，直到第二个 *exp* 的值为止， 其步长为第三个 *exp* 。 更确切的说，一个 **for** 循环看起来是这个样子

```
for v = _e1_, _e2_, _e3_ do _block_ end
```

这等价于代码：

```
do
  local _var_, _limit_, _step_ = tonumber(_e1_), tonumber(_e2_), tonumber(_e3_)
  if not (_var_ and _limit_ and _step_) then error() end
  _var_ = _var_ - _step_
  while true do
    _var_ = _var_ + _step_
    if (_step_ >= 0 and _var_ > _limit_) or (_step_ < 0 and _var_ < _limit_) then
      break
    end
    local v = _var_
    _block_
  end
end
```

注意下面这几点：

- 所有三个控制表达式都只被运算一次， 表达式的计算在循环开始之前。 这些表达式的结果必须是数字。
- `_var_` ， `_limit_` ， 以及 `_step_` 都是一些不可见的变量。 这里给它们起的名字都仅仅用于解释方便。
- 如果第三个表达式（步长）没有给出，会把步长设为 1 。
- 你可以用 **break** 和 **goto** 来退出 **for** 循环。
- 循环变量 `v` 是一个循环内部的局部变量； 如果你需要在循环结束后使用这个值， 在退出循环前把它赋给另一个变量。

通用形式的 **for** 通过一个叫作 迭代器 的函数工作。 每次迭代，迭代器函数都会被调用以产生一个新的值， 当这个值为 **nil** 时，循环停止。 通用形式的 **for** 循环的语法如下：

```
stat ::= **for** namelist **in** explist **do** block **end**
namelist ::= Name {‘**,**’ Name}
```

这样的 **for** 语句

```
for _var_1_, ···, _var_n_ in _explist_ do _block_ end
```

它等价于这样一段代码：

```
do
  local _f_, _s_, _var_ = _explist_
  while true do
    local _var_1_, ···, _var_n_ = _f_(_s_, _var_)
    if _var_1_ == nil then break end
    _var_ = _var_1_
    _block_
  end
end
```

注意以下几点：

- `_explist_` 只会被计算一次。 它返回三个值， 一个 迭代器 函数， 一个 状态， 一个 迭代器的初始值。
- `_f_`， `_s_`， 与 `_var_` 都是不可见的变量。 这里给它们起的名字都只是为了解说方便。
- 你可以使用 **break** 来跳出 **for** 循环。
- 环变量 `_var_i_` 对于循环来说是一个局部变量； 你不可以在 **for** 循环结束后继续使用。 如果你需要保留这些值，那么就在循环跳出或结束前赋值到别的变量里去。

3.3.6 – 函数调用语句

为了允许使用函数的副作用， 函数调用可以被作为一个语句执行：

```
stat ::= functioncall
```

在这种情况下，所有的返回值都被舍弃。 函数调用在 §3.4.10 中解释。

3.3.7 – 局部声明

局部变量可以在语句块中任何地方声明。 声明可以包含一个初始化赋值操作：

```
stat ::= **local** namelist [‘**=**’ explist]
```

如果有初始化值的话，初始化赋值操作的语法和赋值操作一致 （参见 §3.3.3 ）。 若没有初始化值，所有的变量都被初始化为 **nil**。

一个代码块同时也是一个语句块（参见 §3.3.2）， 所以局部变量可以放在代码块中那些显式注明的语句块之外。

局部变量的可见性规则在 §3.5 中解释。

3.4 – 表达式

Lua 中有这些基本表达式：

```
exp ::= prefixexp
exp ::= **nil** | **false** | **true**
exp ::= Numeral
exp ::= LiteralString
exp ::= functiondef
exp ::= tableconstructor
exp ::= ‘**...**’
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | ‘**(**’ exp ‘**)**’
```

数字和字面串在 §3.1 中解释； 变量在 §3.2 中解释； 函数定义在 §3.4.11 中解释； 函数调用在 §3.4.10 中解释； 表的构造在 §3.4.9 中解释。 可变参数的表达式写作三个点（'`...`'），它只能在有可变参数的函数中直接使用；这些在 §3.4.11 中解释。

二元操作符包含有数学运算操作符（参见 §3.4.1）， 位操作符（参见 §3.4.2）， 比较操作符（参见 §3.4.4）， 逻辑操作符（参见 §3.4.5）， 以及连接操作符（参见 §3.4.6）。 一元操作符包括负号（参见 §3.4.1）， 按位非（参见 §3.4.2）， 逻辑非（参见 §3.4.5）， 和取长度操作符（参见 §3.4.7）。

函数调用和可变参数表达式都可以放在多重返回值中。 如果函数调用被当作一条语句（参见 §3.3.6）， 其返回值列表被调整为零个元素，即抛弃所有的返回值。 如果表达式被用于表达式列表的最后（或是唯一的）一个元素， 那么不会做任何调整（除非表达式被括号括起来）。 在其它情况下， Lua 都会把结果调整为一个元素置入表达式列表中， 即保留第一个结果而忽略之后的所有值，或是在没有结果时， 补单个 **nil**。

这里有一些例子：

```
f()                    -- 调整为 0 个结果
g(f(), x)              -- f() 会被调整为一个结果
g(x, f())              -- g 收到 x 以及 f() 返回的所有结果
a,b,c = f(), x         -- f() 被调整为 1 个结果 （c 收到 nil）
a,b = ...              -- a 收到可变参数列表的第一个参数，
                       -- b 收到第二个参数（如果可变参数列表中
                       -- 没有实际的参数，a 和 b 都会收到 nil）

a,b,c = x, f()         -- f() 被调整为 2 个结果
a,b,c = f()            -- f() 被调整为 3 个结果
return f()             -- 返回 f() 的所有返回结果
return ...             -- 返回从可变参数列表中接收到的所有参数parameters
return x,y,f()         -- 返回 x, y, 以及 f() 的所有返回值
{f()}                  -- 用 f() 的所有返回值创建一个列表
{...}                  -- 用可变参数中的所有值创建一个列表
{f(), nil}             -- f() 被调整为一个结果
```

被括号括起来的表达式永远被当作一个值。 所以， `(f(x,y,z))` 即使 `f` 返回多个值， 这个表达式永远是一个单一值。 （`(f(x,y,z))` 的值是 `f` 返回的第一个值。 如果 `f` 不返回值的话，那么它的值就是 **nil** 。）

3.4.1 – 数学运算操作符

Lua 支持下列数学运算操作符：

- `+` : 加法
- `-` : 减法
- `*` : 乘法
- `/` : 浮点除法
- `//` : 向下取整除法
- `%` : 取模
- `^` : 乘方
- `-` : 取负

除了乘方和浮点除法运算， 数学运算按如下方式工作： 如果两个操作数都是整数， 该操作以整数方式操作且结果也将是一个整数。 否则，当两个操作数都是数字或可以被转换为数字的字符串 （参见 §3.4.3）时， 操作数会被转换成两个浮点数， 操作按通常的浮点规则（一般遵循 IEEE 754 标准） 来进行，结果也是一个浮点数。

乘方和浮点除法 （`/`） 总是把操作数转换成浮点数进行，其结果总是浮点数。 乘方使用 ISO C 函数 `pow`， 因此它也可以接受非整数的指数。

向下取整的除法 （`//`） 指做一次除法，并将商圆整到靠近负无穷的一侧， 即对操作数做除法后取 floor 。

取模被定义成除法的余数，其商被圆整到靠近负无穷的一侧（向下取整的除法）。

对于整数数学运算的溢出问题， 这些操作采取的策略是按通常遵循的以 2 为补码的数学运算的 环绕 规则。 （换句话说，它们返回其运算的数学结果对 *2⁶⁴* 取模后的数字。）

3.4.2 – 位操作符

Lua 支持下列位操作符：

- `&` : 按位与
- `|` : 按位或
- `~` : 按位异或
- `>>` : 右移
- `<<` : 左移
- `~` : 按位非

所有的位操作都将操作数先转换为整数 （参见 §3.4.3）， 然后按位操作，其结果是一个整数。

对于右移和左移，均用零来填补空位。 移动的位数若为负，则向反方向位移； 若移动的位数的绝对值大于等于 整数本身的位数，其结果为零 （所有位都被移出）。

3.4.3 – 强制转换

Lua 对一些类型和值的内部表示会在运行时做一些数学转换。 位操作总是将浮点操作数转换成整数。 乘方和浮点除法总是将整数转换为浮点数。 其它数学操作若针对混合操作数 （整数和浮点数）将把整数转换为浮点数； 这一点被称为 通常规则。 C API 同样会按需把整数转换为浮点数以及 把浮点数转换为整数。 此外，字符串连接操作除了字符串，也可以接受数字作为参数。

当操作需要数字时，Lua 还会把字符串转换为数字。

当把一个整数转换为浮点数时， 若整数值恰好可以表示为一个浮点数，那就取那个浮点数。 否则，转换会取最接近的较大值或较小值来表示这个数。 这种转换是不会失败的。

将浮点数转为整数的过程会检查 浮点数能否被准确的表达为一个整数 （即，浮点数是一个整数值且在整数可以表达的区间）。 如果可以，结果就是那个数，否则转换失败。

从字符串到数字的转换过程遵循以下流程： 首先，遵循按 Lua 词法分析器的规则分析语法来转换为对应的 整数或浮点数。 （字符串可以有前置或后置的空格以及一个符号。） 然后，结果数字再按前述规则转换为所需要的类型（浮点或整数）。

从数字转换为字符串使用非指定的人可读的格式。 若想完全控制数字到字符串的转换过程，可以使用字符串库中的 `format` 函数 （参见 `string.format`）。

3.4.4 – 比较操作符

Lua 支持下列比较操作符：

- `==`：等于
- `~=`：不等于
- `<`：小于
- `>`：大于
- `<=`：小于等于
- `>=`：大于等于

这些操作的结果不是 **false** 就是 **true**。

等于操作 （`==`）先比较操作数的类型。 如果类型不同，结果就是 **false**。 否则，继续比较值。 字符串按一般的方式比较。 数字遵循二元操作的规则： 如果两个操作数都是整数， 它们按整数比较； 否则，它们先转换为浮点数，然后再做比较。

表，用户数据，以及线程都按引用比较： 只有两者引用同一个对象时才认为它们相等。 每次你创建一个新对象（一张表，一个用户数据，或一个线程）， 新对象都一定和已有且存在的对象不同。 相同引用的闭包一定相等。 有任何可察觉的差异（不同的行为，不同的定义）一定不等。

你可以通过使用 "eq" 元方法（参见 §2.4） 来改变 Lua 比较表和用户数据时的方式。

等于操作不会将字符串转换为数字，反之亦然。 即， `"0"==0` 结果为 **false**， 且 `t[0]` 与 `t["0"]` 指代着表中的不同项。

`~=` 操作完全等价于 (`==`) 操作的反值。

大小比较操作以以下方式进行。 如果参数都是数字， 它们按二元操作的常规进行。 否则，如果两个参数都是字符串， 它们的值按当前的区域设置来比较。 再则，Lua 就试着调用 "lt" 或是 "le" 元方法 （参见 §2.4）。 `a > b` 的比较被转译为 `b < a`， `a >= b` 被转译为 `b <= a`。

3.4.5 – 逻辑操作符

Lua 中的逻辑操作符有 **and**， **or**，以及 **not**。 和控制结构（参见 §3.3.4）一样， 所有的逻辑操作符把 **false** 和 **nil** 都作为假， 而其它的一切都当作真。

取反操作 **not** 总是返回 **false** 或 **true** 中的一个。 与操作符 **and** 在第一个参数为 **false** 或 **nil** 时 返回这第一个参数； 否则，**and** 返回第二个参数。 或操作符 **or** 在第一个参数不为 **nil** 也不为 **false** 时， 返回这第一个参数，否则返回第二个参数。 **and** 和 **or** 都遵循短路规则； 也就是说，第二个操作数只在需要的时候去求值。 这里有一些例子：

```
10 or 20            --> 10
10 or error()       --> 10
nil or "a"          --> "a"
nil and 10          --> nil
false and error()   --> false
false and nil       --> false
false or nil        --> nil
10 and 20           --> 20
```

（在这本手册中， `-->` 指前面表达式的结果。）

3.4.6 – 字符串连接

Lua 中字符串的连接操作符写作两个点（'`..`'）。 如果两个操作数都是字符串或都是数字， 连接操作将以 §3.4.3 中提到的规则把其转换为字符串。 否则，会调用元方法 `__concat` （参见 §2.4）。

3.4.7 – 取长度操作符

取长度操作符写作一元前置符 `#`。 字符串的长度是它的字节数（就是以一个字符一个字节计算的字符串长度）。

程序可以通过 `__len` 元方法（参见 §2.4） 来修改对字符串类型外的任何值的取长度操作行为。

如果 `__len` 元方法没有给出， 表 `t` 的长度只在表是一个 序列 时有定义。 序列指表的正数键集等于 *{1..n}* ， 其中 *n* 是一个非负整数。 在这种情况下，*n* 是表的长度。 注意这样的表

```
{10, 20, nil, 40}
```

不是一个序列，因为它有键 `4` 却没有键 `3`。 （因此，该表的正整数键集不等于 *{1..n}* 集合，故而就不存在 *n*。） 注意，一张表是否是一个序列和它的非数字键无关。

3.4.8 – 优先级

Lua 中操作符的优先级写在下表中，从低到高优先级排序：

```
or
and
<     >     <=    >=    ~=    ==
|
~
&
<<    >>
..
+     -
*     /     //    %
unary operators (not   #     -     ~)
^
```

通常， 你可以用括号来改变运算次序。 连接操作符 ('`..`') 和乘方操作 ('`^`') 是从右至左的。其它所有的操作都是从左至右。

3.4.9 – 表构建

表构造子是一个构造表的表达式。 每次构造子被执行，都会构造出一张新的表。 构造子可以被用来构造一张空表， 也可以用来构造一张表并初始化其中的一些域。 一般的构造子的语法如下

```
tableconstructor ::= ‘**{**’ [fieldlist] ‘**}**’
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= ‘**[**’ exp ‘**]**’ ‘**=**’ exp | Name ‘**=**’ exp | exp
fieldsep ::= ‘**,**’ | ‘**;**’
```

每个形如 `[exp1] = exp2` 的域向表中增加新的一项， 其键为 `exp1` 而值为 `exp2` 。 形如 `name = exp` 的域等价于 `["name"] = exp` 。 最后，形如 `exp` 的域等价于 `[i] = exp` ， 这里的 `i` 是一个从 1 开始不断增长的数字。 这这个格式中的其它域不会破坏其记数。 举个例子：

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

等价于

```
do
  local t = {}
  t[f(1)] = g
  t[1] = "x"         -- 1st exp
  t[2] = "y"         -- 2nd exp
  t.x = 1            -- t["x"] = 1
  t[3] = f(x)        -- 3rd exp
  t[30] = 23
  t[4] = 45          -- 4th exp
  a = t
end
```

构造子中赋值的次序未定义。 （次序问题只会对那些键重复时的情况有影响。）

如果表单中最后一个域的形式是 `exp` ， 而且其表达式是一个函数调用或者是一个可变参数， 那么这个表达式所有的返回值将依次进入列表 （参见 §3.4.10）。

初始化域表可以在最后多一个分割符， 这样设计可以方便由机器生成代码。

3.4.10 – 函数调用

Lua 中的函数调用的语法如下：

```
functioncall ::= prefixexp args
```

函数调用时， 第一步，prefixexp 和 args 先被求值。 如果 prefixexp 的值的类型是 *function*，那么这个函数就被用给出的参数调用。 否则 prefixexp 的元方法 "call" 就被调用， 第一个参数是 prefixexp 的值， 接下来的是原来的调用参数 （参见 §2.4）。

这样的形式

```
functioncall ::= prefixexp '**:**' Name args
```

可以用来调用 "方法"。 这是 Lua 支持的一种语法糖。 像 `v:name(args)` 这个样子， 被解释成 `v.name(v,args)`， 这里的 `v` 只会被求值一次。

参数的语法如下：

```
args ::= '**(**' [explist] '**)**'
args ::= tableconstructor
args ::= LiteralString
```

所有参数的表达式求值都在函数调用之前。 这样的调用形式 `f{fields}` 是一种语法糖用于表示 `f({fields})`； 这里指参数列表是一个新创建出来的列表。 而这样的形式 `f'_string_'` （或是 `f"_string_"` 亦或是 `f[[_string_]]`） 也是一种语法糖，用于表示 `f('_string_')`； 此时的参数列表是一个单独的字符串。

`return _functioncall_` 这样的调用形式将触发一次 尾调用。 Lua 实现了 完全尾调用（或称为 完全尾递归）： 在尾调用中， 被调用的函数重用调用它的函数的堆栈项。 因此，对于程序执行的嵌套尾调用的层数是没有限制的。 然而，尾调用将删除调用它的函数的任何调试信息。 注意，尾调用只发生在特定的语法下， 仅当 **return** 只有单一函数调用作为参数时才发生尾调用； 这种语法使得调用函数的所有结果可以完整地返回。 因此，下面这些例子都不是尾调用：

```
return (f(x))        -- 返回值被调整为一个
return 2 * f(x)
return x, f(x)       -- 追加若干返回值
f(x); return         -- 返回值全部被舍弃
return x or f(x)     -- 返回值被调整为一个
```

3.4.11 – 函数定义

函数定义的语法如下：

```
functiondef ::= **function** funcbody
funcbody ::= '**(**' [parlist] '**)**' block **end**
```

另外定义了一些语法糖简化函数定义的写法：

```
stat ::= **function** funcname funcbody
stat ::= **local** **function** Name funcbody
funcname ::= Name {‘**.**’ Name} [‘**:**’ Name]
```

该语句

```
function f () _body_ end
```

被转译成

```
f = function () _body_ end
```

该语句

```
function t.a.b.c.f () _body_ end
```

被转译成

```
t.a.b.c.f = function () _body_ end
```

该语句

```
local function f () _body_ end
```

被转译成

```
local f; f = function () _body_ end
```

而不是

```
local f = function () _body_ end
```

（这个差别只在函数体内需要引用 `f` 时才有。）

一个函数定义是一个可执行的表达式， 执行结果是一个类型为 *function* 的值。 当 Lua 预编译一个代码块时， 代码块作为一个函数，整个函数体也就被预编译了。 那么，无论何时 Lua 执行了函数定义， 这个函数本身就进行了 实例化（或者说是 关闭了）。 这个函数的实例（或者说是 闭包）是表达式的最终值。

形参被看作是一些局部变量， 它们将由实参的值来初始化：

```
parlist ::= namelist [‘**,**’ ‘**...**’] | ‘**...**’
```

当一个函数被调用， 如果函数并非一个 可变参数函数， 即在形参列表的末尾注明三个点 ('`...`')， 那么实参列表就会被调整到形参列表的长度。 变长参数函数不会调整实参列表； 取而代之的是，它将把所有额外的参数放在一起通过 变长参数表达式传递给函数， 其写法依旧是三个点。 这个表达式的值是一串实参值的列表， 看起来就跟一个可以返回多个结果的函数一样。 如果一个变长参数表达式放在另一个表达式中使用， 或是放在另一串表达式的中间， 那么它的返回值就会被调整为单个值。 若这个表达式放在了一系列表达式的最后一个，就不会做调整了 （除非这最后一个参数被括号给括了起来）。

我们先做如下定义，然后再来看一个例子：

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

下面看看实参到形参数以及可变长参数的映射关系：

```
CALL            PARAMETERS

f(3)             a=3, b=nil
f(3, 4)          a=3, b=4
f(3, 4, 5)       a=3, b=4
f(r(), 10)       a=1, b=10
f(r())           a=1, b=2

g(3)             a=3, b=nil, ... -->  (nothing)
g(3, 4)          a=3, b=4,   ... -->  (nothing)
g(3, 4, 5, 8)    a=3, b=4,   ... -->  5  8
g(5, r())        a=5, b=1,   ... -->  2  3
```

结果由 **return** 来返回（参见 §3.3.4）。 如果执行到函数末尾依旧没有遇到任何 **return** 语句， 函数就不会返回任何结果。

关于函数可返回值的数量限制和系统有关。 这个限制一定大于 1000 。

冒号 语法可以用来定义 方法， 就是说，函数可以有一个隐式的形参 `self` 。 因此，如下语句

```
function t.a.b.c:f (_params_) _body_ end
```

是这样一种写法的语法糖

```
t.a.b.c.f = function (self, _params_) _body_ end
```

3.5 – 可见性规则

Lua 语言有词法作用范围。 变量的作用范围开始于声明它们之后的第一个语句段， 结束于包含这个声明的最内层语句块的最后一个非空语句。 看下面这些例子：

```
x = 10                -- 全局变量
do                    -- 新的语句块
  local x = x         -- 新的一个 'x', 它的值现在是 10
  print(x)            --> 10
  x = x+1
  do                  -- 另一个语句块
    local x = x+1     -- 又一个 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10 （取到的是全局的那一个）
```

注意这里，类似 `local x = x` 这样的声明， 新的 `x` 正在被声明，但是还没有进入它的作用范围， 所以第二个 `x` 指向的是外面一层的变量。

因为有这样一个词法作用范围的规则， 局部变量可以被在它的作用范围内定义的函数自由使用。 当一个局部变量被内层的函数中使用的时候， 它被内层函数称作 上值，或是 外部局部变量。

注意，每次执行到一个 **local** 语句都会定义出一个新的局部变量。 看看这样一个例子：

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
end
```

这个循环创建了十个闭包（这指十个匿名函数的实例）。 这些闭包中的每一个都使用了不同的 `y` 变量， 而它们又共享了同一份 `x` 。

4 – 编程接口

这个部分描述了 Lua 的 C API ， 也就是宿主程序跟 Lua 通讯用的一组 C 函数。 所有的 API 函数按相关的类型以及常量都声明在头文件 `lua.h` 中。

虽然我们说的是“函数”， 但一部分简单的 API 是以宏的形式提供的。 除非另有说明， 所有的这些宏都只使用它们的参数一次 （除了第一个参数，那一定是 Lua 状态）， 因此你不需担心这些宏的展开会引起一些副作用。

C 库中所有的 Lua API 函数都不去检查参数是否相容及有效。 然而，你可以在编译 Lua 时加上打开一个宏开关 `LUA_USE_APICHECK` 来改变这个行为。

4.1 – 栈

Lua 使用一个 虚拟栈 来和 C 互传值。 栈上的的每个元素都是一个 Lua 值 （**nil**，数字，字符串，等等）。

无论何时 Lua 调用 C，被调用的函数都得到一个新的栈， 这个栈独立于 C 函数本身的栈，也独立于之前的 Lua 栈。 它里面包含了 Lua 传递给 C 函数的所有参数， 而 C 函数则把要返回的结果放入这个栈以返回给调用者 （参见 `lua_CFunction` ）。

方便起见， 所有针对栈的 API 查询操作都不严格遵循栈的操作规则。 而是可以用一个 索引 来指向栈上的任何元素： 正的索引指的是栈上的绝对位置（从1开始） ； 负的索引则指从栈顶开始的偏移量。 展开来说，如果堆栈有 *n* 个元素， 那么索引 1 表示第一个元素 （也就是最先被压栈的元素） 而索引 *n* 则指最后一个元素 ; 索引 -1 也是指最后一个元素 （即栈顶的元素）， 索引 -*n* 是指第一个元素。

4.2 – 栈大小

当你使用 Lua API 时， 就有责任保证做恰当的调用。 特别需要注意的是， 你有责任控制不要堆栈溢出。 你可以使用 `lua_checkstack` 这个函数来扩大可用堆栈的尺寸。

无论何时 Lua 调用 C ， 它都只保证至少有 `LUA_MINSTACK` 这么多的堆栈空间可以使用。 `LUA_MINSTACK` 一般被定义为 20 ， 因此，只要你不是不断的把数据压栈， 通常你不用关心堆栈大小。

当你调用一个 Lua 函数却没有指定要接收多少个返回值时 （参见 `lua_call` ）， Lua 可以保证栈一定有足够的空间来接收所有的返回值， 但不保证此外留有额外的空间。 因此，在做了一次这样的调用后，如果你需要继续压栈， 则需要使用 `lua_checkstack` 。

4.3 – 有效索引与可接受索引

API 中的函数若需要传入栈索引，这个索引必须是 有效索引 或是 可接受索引。

有效索引 指引用栈内真实位置的索引；即在 1 到栈顶之间的位置（ `1 ≤ abs(index) ≤ top` ）。 通常，一个可能修改该位置的值的函数需要传入有效索引。

除非另有说明， 任何可以接受有效索引的函数同时也接受 伪索引。 伪索引指代一些可以被 C code 访问得到 Lua 值，而它们又不在栈内。 这用于访问注册表以及 C 函数的上值（参见 §4.4）。

对于那些只是需要栈中的值（例如查询函数） 而不需要指定一个栈位置的函数， 可以用一个可接受的索引去调用它们。 可接受索引 不仅可以是任何包括伪索引在内的有效索引， 还可以是任何超过栈顶但落在为栈分配出来的空间内的正索引。 （注意 0 永远都不是一个可接受索引。） 除非另有说明，API 里的函数都接受可接受索引。

允许可接受索引是为了避免对栈顶以外的查询时做额外的检查。 例如，C 函数可以直接查询传给它的第三个参数， 而不用先检查是不是有第三个参数， 即不需要检查 3 是不是一个有效索引。

对于那些以可接受索引调用的函数， 无效索引被看作包含了一个虚拟类型 `LUA_TNONE` 的值， 这个值的行为和 nil 一致。

4.4 – C 闭包

当 C 函数被创建出来， 我们有可能会把一些值关联在一起， 也就是创建一个 *C 闭包* （参见 `lua_pushcclosure` ）； 这些被关联起来的值被叫做 上值 ， 它们可以在函数被调用的时候访问的到。

无论何时去调用 C 函数， 函数的上值都可以用伪索引定位。 我们可以用 `lua_upvalueindex` 这个宏来生成这些伪索引。 第一个关联到函数的值放在 `lua_upvalueindex(1)` 位置处， 依此类推。 使用 `lua_upvalueindex(_n_)` 时， 若 *n* 大于当前函数的总上值个数 （但不可以大于 256）会产生一个可接受的但无效的索引。

4.5 – 注册表

Lua 提供了一个 注册表， 这是一个预定义出来的表， 可以用来保存任何 C 代码想保存的 Lua 值。 这个表可以用有效伪索引 `LUA_REGISTRYINDEX` 来定位。 任何 C 库都可以在这张表里保存数据， 为了防止冲突，你需要特别小心的选择键名。 一般的用法是，你可以用一个包含你的库名的字符串做为键名， 或者取你自己 C 对象的地址，以轻量用户数据的形式做键， 还可以用你的代码创建出来的任意 Lua 对象做键。 关于变量名，字符串键名中以下划线加大写字母的名字被 Lua 保留。

注册表中的整数键用于引用机制（参见 `luaL_ref`）， 以及一些预定义的值。 因此，整数键不要用于别的目的。

当你创建了一个新的 Lua 状态机， 其中的注册表内就预定义好了几个值。 这些预定义值可以用整数索引到， 这些整数以常数形式定义在 `lua.h` 中。 有下列常数：

- **`LUA_RIDX_MAINTHREAD`**：注册表中这个索引下是状态机的主线程。 （主线程和状态机同时被创建出来。）
- **`LUA_RIDX_GLOBALS`**：注册表的这个索引下是全局环境。

4.6 – C 中的错误处理

在内部实现中，Lua 使用了 C 的 `longjmp` 机制来处理错误。 （如果你使用 C++ 编译，Lua 将换成异常； 细节请在源代码中搜索 `LUAI_THROW`。） 当 Lua 碰到任何错误 （比如内存分配错误、 类型错误、 语法错误、 还有运行时错误） 它都会 抛出一个错误出去； 也就是调用一次长跳转。 在 保护环境 下， Lua 使用 `setjmp` 来设置一个恢复点； 任何发生的错误都会跳转到最近的一个恢复点。

如果错误发生在保护环境之外， Lua 会先调用 *panic* 函数 （参见 `lua_atpanic`） 然后调用 `abort` 来退出宿主程序。 你的 panic 函数只要不返回 （例如：长跳转到你在 Lua 外你自己设置的恢复点） 就可以不退出程序。

panic 函数以错误消息处理器（参见 §2.3）的方式运行； 错误消息在栈顶。 不同的是，它不保证栈空间。 做任何压栈操作前，panic 函数都必须先检查是否有足够的空间 （参见 §4.2）。

大多数 API 函数都有可能抛出错误， 例如在内存分配错误时就会抛出。 每个函数的文档都会注明它是否可能抛出错误。

在 C 函数内部，你可以通过调用 `lua_error` 来抛出错误。

4.7 – C 中的让出处理

Lua 内部使用 C 的 `longjmp` 机制让出一个协程。 因此，如果一个 C 函数 `foo` 调用了一个 API 函数， 而这个 API 函数让出了（直接或间接调用了让出函数）。 由于 `longjmp` 会移除 C 栈的栈帧， Lua 就无法返回到 `foo` 里了。

为了回避这类问题， 碰到 API 调用中调用让出时，除了那些抛出错误的 API 外，还提供了三个函数： `lua_yieldk`， `lua_callk`，和 `lua_pcallk` 。 它们在让出发生时，可以从传入的延续函数 （名为 `k` 的参数）继续运行。

我们需要预设一些术语来解释延续点。 对于从 Lua 中调用的 C 函数，我们称之为 原函数。 从这个原函数中调用的上面所述的三个 C API 函数我们称之为 被调函数。 被调函数可以使当前线程让出。 （让出发生在被调函数是 `lua_yieldk`， 或传入 `lua_callk` 或 `lua_pcallk` 的函数调用了让出时。）

假设正在运行的线程在执行被调函数时让出。 当再次延续这条线程，它希望继续被调函数的运行。 然而，被调函数不可能返回到原函数中。 这是因为之前的让出操作破坏了 C 栈的栈帧。 作为替代品，Lua 调用那个作为被调函数参数给出的 延续函数 。 正如其名，延续函数将延续原函数的任务。

下面的函数会做一个说明：

```
int original_function (lua_State *L) {
  ...     /* code 1 */
  status = lua_pcall(L, n, m, h);  /* calls Lua */
  ...     /* code 2 */
}
```

现在我们想允许被 `lua_pcall` 运行的 Lua 代码让出。 首先，我们把函数改写成这个样子：

```
int k (lua_State *L, int status, lua_KContext ctx) {
  ...  /* code 2 */
}

int original_function (lua_State *L) {
  ...     /* code 1 */
  return k(L, lua_pcall(L, n, m, h), ctx);
}
```

上面的代码中，新函数 `k` 就是一个 延续函数 （函数类型为 `lua_KFunction`）。 它的工作就是原函数中调用 `lua_pcall` 之后做的那些事情。 现在我们必须通知 Lua 说，你必须在被 `lua_pcall` 执行的 Lua 代码发生过中断（错误或让出）后， 还得继续调用 `k` 。 所以我们还得继续改写这段代码，把 `lua_pcall` 替换成 `lua_pcallk` ：

```
int original_function (lua_State *L) {
  ...     /* code 1 */
  return k(L, lua_pcallk(L, n, m, h, ctx2, k), ctx1);
}
```

注意这里那个额外的显式的对延续函数的调用： Lua 仅在需要时，这可能是由错误导致的也可能是发生了让出而需要继续运行，才会调用延续函数。 如果没有发生过任何让出，调用的函数正常返回， 那么 `lua_pcallk` （以及 `lua_callk`）也会正常返回。 （当然，这个例子中你也可以不在之后调用延续函数， 而是在原函数的调用后直接写上需要做的工作。）

除了 Lua 状态，延续函数还有两个参数： 一个是调用最后的状态码，另一个一开始由 `lua_pcallk` 传入的上下文 （ `ctx` ）。 （Lua 本身不使用这个值；它仅仅从原函数转发这个值给延续函数。） 对于 `lua_pcallk` 而言， 状态码和 `lua_pcallk` 本应返回值相同，区别仅在于发生过让出后才执行完时，状态码为 `LUA_YIELD` （而不是 `LUA_OK`）。 对于

`lua_yieldk` 和 `lua_callk` 而言， 调用延续函数传入的状态码一定是 `LUA_YIELD` 。 （对这两个函数，Lua 不会因任何错误而调用延续函数。 因为它们并不处理错误。） 同样，当你使用 `lua_callk` 时， 你应该用 `LUA_OK` 作为状态码来调用延续函数。 （对于 `lua_yieldk`， 几乎没有什么地方需要直接调用延续函数， 因为 `lua_yieldk` 本身并不会返回。）

Lua 会把延续函数看作原函数。 延续函数将接收到和原函数相同的 Lua 栈，其接收到的 lua 状态也和 被调函数若返回后应该有的状态一致。 （例如， `lua_callk` 调用之后， 栈中之前压入的函数和调用参数都被调用产生的返回值所替代。） 这时也有相同的上值。 等到它返回的时候，Lua 会将其看待成原函数的返回去操作。

4.8 – 函数和类型

这里按字母次序列出了所有 C API 中的函数和类型。 每个函数都有一个这样的提示： [-o, +p, *x*]

对于第一个域， `o` ， 指的是该函数会从栈上弹出多少个元素。 第二个域， `p` ， 指该函数会将多少个元素压栈。 （所有函数都会在弹出参数后再把结果压栈。） `x|y` 这种形式的域表示该函数根据具体情况可能压入（或弹出） `x` 或 `y` 个元素； 问号 ' `?` ' 表示 我们无法仅通过参数来了解该函数会弹出/压入多少元素 （比如，数量取决于栈上有些什么）。 第三个域， `x` ， 解释了该函数是否会抛出错误： ' `-` ' 表示该函数绝对不会抛出错误； ' `e` ' 表示该函数可能抛出错误； ' `v` ' 表示该函数可能抛出有意义的错误。

`lua_absindex`

[-0, +0, –]

```
int lua_absindex (lua_State *L, int idx);
```

将一个可接受的索引 `idx` 转换为绝对索引 （即，一个不依赖栈顶在哪的值）。

`lua_Alloc`

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

Lua 状态机中使用的内存分配器函数的类型。 内存分配函数必须提供一个功能类似于 `realloc` 但又不完全相同的函数。 它的参数有 `ud` ， 一个由 `lua_newstate` 传给它的指针； `ptr` ， 一个指向已分配出来/将被重新分配/要释放的内存块指针； `osize` ， 内存块原来的尺寸或是关于什么将被分配出来的代码； `nsize` ， 新内存块的尺寸。

如果 `ptr` 不是 `NULL` ， `osize` 是 `ptr` 指向的内存块的尺寸， 即这个内存块当初被分配或重分配的尺寸。

如果 `ptr` 是 `NULL` ， `osize` 是 Lua 即将分配对象类型的编码。 当（且仅当）Lua 创建一个对应类型的新对象时， `osize` 是 `LUA_TSTRING` ， `LUA_TTABLE` ， `LUA_TFUNCTION` ， `LUA_TUSERDATA` ， 或 `LUA_TTHREAD` 中的一个。 若 `osize` 是其它类型，Lua 将为其它东西分配内存。

Lua 假定分配器函数会遵循以下行为：

当 `nsize` 是零时， 分配器必须和 `free` 行为类似并返回 `NULL` 。

当 `nsize` 不是零时， 分配器必须和 `realloc` 行为类似。 如果分配器无法完成请求，返回 `NULL` 。 Lua 假定在 `osize >= nsize` 成立的条件下， 分配器绝不会失败。

这里有一个简单的分配器函数的实现。 这个实现被放在补充库中，供 `luaL_newstate` 使用。

```
static void *l_alloc (void *ud, void *ptr, size_t osize,
                                           size_t nsize) {
  (void)ud;  (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

注意，标准 C 能确保 `free(NULL)` 没有副作用， 且 `realloc(NULL,size)` 等价于 `malloc(size)` 。 这段代码假定 `realloc` 在缩小块长度时不会失败。 （虽然标准 C 没有对此行为做出保证，但这看起来是一个安全的假定。）

`lua_arith`

[-(2|1), +1, *e*]

```
void lua_arith (lua_State *L, int op);
```

对栈顶的两个值（或者一个，比如取反）做一次数学或位操作。 其中，栈顶的那个值是第二个操作数。 它会弹出压入的值，并把结果放在栈顶。 这个函数遵循 Lua 对应的操作符运算规则 （即有可能触发元方法）。

`op` 的值必须是下列常量中的一个：

- **`LUA_OPADD`**：加法 (`+`)
- **`LUA_OPSUB`**：减法 (`-`)
- **`LUA_OPMUL`**：乘法 (`*`)
- **`LUA_OPDIV`**：浮点除法 (`/`)

- `LUA_OPIDIV`：向下取整的除法 (`//`)
- `LUA_OPMOD`：取模 (`%`)
- `LUA_OPPOW`：乘方 (`^`)
- `LUA_OPUNM`：取负 (一元 `-`)
- `LUA_OPBNOT`：按位取反 (`~`)
- `LUA_OPBAND`：按位与 (`&`)
- `LUA_OPBOR`：按位或 (`|`)
- `LUA_OPBXOR`：按位异或 (`~`)
- `LUA_OPSHL`：左移 (`<<`)
- `LUA_OPSHR`：右移 (`>>`)

`lua_atpanic`

[-0, +0, –]

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

设置一个新的 panic 函数，并返回之前设置的那个。（参见 §4.6）。

`lua_call`

[-(nargs+1), +nresults, *e*]

```
void lua_call (lua_State *L, int nargs, int nresults);
```

调用一个函数。

要调用一个函数请遵循以下协议：首先，要调用的函数应该被压入栈；接着，把需要传递给这个函数的参数按正序压栈；这是指第一个参数首先压栈。最后调用一下 `lua_call`；`nargs` 是你压入栈的参数个数。当函数调用完毕后，所有的参数以及函数本身都会出栈。而函数的返回值这时则被压栈。返回值的个数将被调整为 `nresults` 个，除非 `nresults` 被设置成 `LUA_MULTRET`。在这种情况下，所有的返回值都被压入堆栈中。Lua 会保证返回值都放入栈空间中。函数返回值将按正序压栈（第一个返回值首先压栈），因此在调用结束后，最后一个返回值将被放在栈顶。

被调用函数内发生的错误将（通过 `longjmp` ）一直上抛。

下面的例子中，这行 Lua 代码等价于在宿主程序中用 C 代码做一些工作：

```
     a = f("how", t.x, 14)
```

这里是 C 里的代码：

```
     lua_getglobal(L, "f");                  /* function to be called */
     lua_pushliteral(L, "how");                       /* 1st argument */
     lua_getglobal(L, "t");                    /* table to be indexed */
     lua_getfield(L, -1, "x");        /* push result of t.x (2nd arg) */
     lua_remove(L, -2);                  /* remove 't' from the stack */
     lua_pushinteger(L, 14);                          /* 3rd argument */
     lua_call(L, 3, 1);     /* call 'f' with 3 arguments and 1 result */
     lua_setglobal(L, "a");                         /* set global 'a' */
```

注意上面这段代码是 平衡 的： 到了最后，堆栈恢复成原有的配置。 这是一种良好的编程习惯。

`lua_callk`

[-(nargs + 1), +nresults, *e*]

```
void lua_callk (lua_State *L,
                int nargs,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```

这个函数的行为和 `lua_call` 完全一致，只不过它还允许被调用的函数让出 （参见 §4.7）。

`lua_CFunction`

```
typedef int (*lua_CFunction) (lua_State *L);
```

C 函数的类型。

为了正确的和 Lua 通讯， C 函数必须使用下列协议。 这个协议定义了参数以及返回值传递方法： C 函数通过 Lua 中的栈来接受参数， 参数以正序入栈（第一个参数首先入栈）。 因此，当函数开始的时候， `lua_gettop(L)` 可以返回函数收到的参数个数。 第一个参数（如果有的话）在索引 1 的地方， 而最后一个参数在索引 `lua_gettop(L)` 处。 当需要向 Lua 返回值的时候， C 函数只需要把它们以正序压到堆栈上（第一个返回值最先压入）， 然后返回这些返回值的个数。 在这些返回值之下的，堆栈上的东西都会被 Lua 丢掉。 和 Lua 函数一样，从 Lua 中调用 C 函数也可以有很多返回值。

下面这个例子中的函数将接收若干数字参数，并返回它们的平均数与和：

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);    /* 参数的个数 */
  lua_Number sum = 0.0;
  int i;
  for (i = 1; i <= n; i++) {
    if (!lua_isnumber(L, i)) {
      lua_pushliteral(L, "incorrect argument");
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
  lua_pushnumber(L, sum/n);        /* 第一个返回值 */
  lua_pushnumber(L, sum);         /* 第二个返回值 */
  return 2;                   /* 返回值的个数 */
}
```

lua_checkstack

[-0, +0, –]

```
int lua_checkstack (lua_State *L, int n);
```

确保堆栈上至少有 `n` 个额外空位。 如果不能把堆栈扩展到相应的尺寸，函数返回假。 失败的原因包括将把栈扩展到比固定最大尺寸还大 （至少是几千个元素）或分配内存失败。 这个函数永远不会缩小堆栈； 如果堆栈已经比需要的大了，那么就保持原样。

lua_close

[-0, +0, –]

```
void lua_close (lua_State *L);
```

销毁指定 Lua 状态机中的所有对象 （如果有垃圾收集相关的元方法的话，会调用它们）， 并且释放状态机中使用的所有动态内存。 在一些平台上，你可以不必调用这个函数， 因为当宿主程序结束的时候，所有的资源就自然被释放掉了。 另一方面，长期运行的程序，比如一个后台程序或是一个网站服务器， 会创建出多个 Lua 状态机。那么就应该在不需要时赶紧关闭它们。

lua_compare

[-0, +0, *e*]

```
int lua_compare (lua_State *L, int index1, int index2, int op);
```

比较两个 Lua 值。 当索引 `index1` 处的值通过 `op` 和索引 `index2` 处的值做比较后条件满足，函数返回 1 。 这个函数遵循 Lua 对应的操作规则（即有可能触发元方法）。 反之，函数返回 0。 当任何一个索引无效时，函数也会返回 0 。

`op` 值必须是下列常量中的一个：

- **`LUA_OPEQ`**：相等比较 (`==`)
- **`LUA_OPLT`**：小于比较 (`<`)
- **`LUA_OPLE`**：小于等于比较 (`<=`)

`lua_concat`

[-n, +1, *e*]

```
void lua_concat (lua_State *L, int n);
```

连接栈顶的 `n` 个值， 然后将这些值出栈，并把结果放在栈顶。 如果 `n` 为 1 ，结果就是那个值放在栈上（即，函数什么都不做） ； 如果 `n` 为 0 ，结果是一个空串。 连接依照 Lua 中通常语义完成（参见 §3.4.6 ） 。

`lua_copy`

[-0, +0, –]

```
void lua_copy (lua_State *L, int fromidx, int toidx);
```

从索引 `fromidx` 处复制一个值到一个有效索引 `toidx` 处，覆盖那里的原有值。 不会影响其它位置的值。

`lua_createtable`

[-0, +1, *e*]

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

创建一张新的空表压栈。 参数 `narr` 建议了这张表作为序列使用时会有多少个元素； 参数 `nrec` 建议了这张表可能拥有多少序列之外的元素。 Lua 会使用这些建议来预分配这张新表。 如果你知道这张表用途的更多信息，预分配可以提高性能。 否则，你可以使用函数 `lua_newtable` 。

`lua_dump`

[-0, +0, *e*]

```
int lua_dump (lua_State *L,
                        lua_Writer writer,
                        void *data,
                        int strip);
```

把函数导出成二进制代码块 。 函数接收栈顶的 Lua 函数做参数， 然后生成它的二进制代码块。 若被导出的东西被再次加载， 加载的结果就相当于原来的函数。 当它在产生代码块的时候， `lua_dump` 通过调用函数 `writer` （参见 `lua_Writer` ） 来写入数据，后面的 `data` 参数会被传入 `writer` 。

如果 `strip` 为真， 二进制代码块将不包含该函数的调试信息。

最后一次由 `writer` 的返回值将作为这个函数的返回值返回； 0 表示没有错误。

该函数不会把 Lua 函数弹出堆栈。

`lua_error`

[-1, +0, *v*]

```
int lua_error (lua_State *L);
```

以栈顶的值作为错误对象，抛出一个 Lua 错误。 这个函数将做一次长跳转，所以一定不会返回 （参见 `luaL_error` ）。

`lua_gc`

[-0, +0, *e*]

```
int lua_gc (lua_State *L, int what, int data);
```

控制垃圾收集器。

这个函数根据其参数 `what` 发起几种不同的任务：

- **`LUA_GCSTOP`**：停止垃圾收集器。
- **`LUA_GCRESTART`**：重启垃圾收集器。
- **`LUA_GCCOLLECT`**：发起一次完整的垃圾收集循环。
- **`LUA_GCCOUNT`**：返回 Lua 使用的内存总量（以 K 字节为单位）。
- **`LUA_GCCOUNTB`**：返回当前内存使用量除以 1024 的余数。
- **`LUA_GCSTEP`**：发起一步增量垃圾收集。
- **`LUA_GCSETPAUSE`**：把 `data` 设为 垃圾收集器间歇率 （参见 §2.5），并返回之前设置的值。
- **`LUA_GCSETSTEPMUL`**：把 `data` 设为 垃圾收集器步进倍率 （参见 §2.5）， 并返回之前设置

的值。

- `LUA_GCISRUNNING`：返回收集器是否在运行（即没有停止）。

关于这些选项的细节，参见 `collectgarbage` 。

`lua_getallocf`

[-0, +0, –]

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

返回给定状态机的内存分配器函数。 如果 `ud` 不是 `NULL` ， Lua 把设置内存分配函数时设置的那个指针置入 `*ud` 。

`lua_getfield`

[-0, +1, *e*]

```
int lua_getfield (lua_State *L, int index, const char *k);
```

把 `t[k]` 的值压栈， 这里的 `t` 是索引指向的值。 在 Lua 中，这个函数可能触发对应 "index" 事件对应的元方法 （参见 §2.4 ）。

函数将返回压入值的类型。

`lua_getextraspace`

[-0, +0, –]

```
void *lua_getextraspace (lua_State *L);
```

返回一个 Lua 状态机中关联的内存块指针。 程序可以把这块内存用于任何用途；而 Lua 不会使用它。

每一个新线程都会携带一块内存， 初始化为主线程的这块内存的副本。

默认配置下，这块内存的大小为空指针的大小。 不过你可以重新编译 Lua 设定这块内存不同的大小。 （参见 `luaconf.h` 中的 `LUA_EXTRASPACE` 。）

`lua_getglobal`

[-0, +1, *e*]

```
int lua_getglobal (lua_State *L, const char *name);
```

把全局变量 <name>name</name> 里的值压栈，返回该值的类型。

lua_geti

[-0, +1, *e*]

```
int lua_geti (lua_State *L, int index, lua_Integer i);
```

把 `t[i]` 的值压栈， 这里的 `t` 指给定的索引指代的值。 和在 Lua 里一样，这个函数可能会触发 "index" 事件的元方法 （参见 §2.4）。

返回压入值的类型。

lua_getmetatable

[-0, +(0|1), –]

```
int lua_getmetatable (lua_State *L, int index);
```

如果该索引处的值有元表，则将其元表压栈，返回 1 。 否则不会将任何东西入栈，返回 0 。

lua_gettable

[-1, +1, *e*]

```
int lua_gettable (lua_State *L, int index);
```

把 `t[k]` 的值压栈， 这里的 `t` 是指索引指向的值， 而 `k` 则是栈顶放的值。

这个函数会弹出堆栈上的键，把结果放在栈上相同位置。 和在 Lua 中一样， 这个函数可能触发对应 "index" 事件的元方法 （参见 §2.4 ）。

返回压入值的类型。

lua_gettop

[-0, +0, –]

```
int lua_gettop (lua_State *L);
```

返回栈顶元素的索引。 因为索引是从 1 开始编号的， 所以这个结果等于栈上的元素个数； 特别指出，0 表示栈为空。

`lua_getuservalue`

[-0, +1, –]

```
int lua_getuservalue (lua_State *L, int index);
```

将给定索引处的用户数据所关联的 Lua 值压栈。

返回压入值的类型。

`lua_insert`

[-1, +1, –]

```
void lua_insert (lua_State *L, int index);
```

把栈顶元素移动到指定的有效索引处， 依次移动这个索引之上的元素。 不要用伪索引来调用这个函数， 因为伪索引没有真正指向栈上的位置。

`lua_Integer`

```
typedef ... lua_Integer;
```

Lua 中的整数类型。

缺省时，这个就是 `long long`， （通常是一个 64 位以二为补码的整数）， 也可以修改它为 `long` 或 `int` （通常是一个 32 位以二为补码的整数）。 （参见 `luaconf.h` 中的 `LUA_INT` 。）

Lua 定义了两个常量： `LUA_MININTEGER` 和 `LUA_MAXINTEGER` 来表示这个类型可以表示的最小和最大值。

`lua_isboolean`

[-0, +0, –]

```
int lua_isboolean (lua_State *L, int index);
```

当给定索引的值是一个布尔量时，返回 1 ， 否则返回 0 。

`lua_iscfunction`

[-0, +0, –]

```
int lua_iscfunction (lua_State *L, int index);
```

当给定索引的值是一个 C 函数时，返回 1 ，否则返回 0 。

lua_isfunction

[-0, +0, –]

```
int lua_isfunction (lua_State *L, int index);
```

当给定索引的值是一个函数（ C 或 Lua 函数均可）时，返回 1 ，否则返回 0 。

lua_isinteger

[-0, +0, –]

```
int lua_isinteger (lua_State *L, int index);
```

当给定索引的值是一个整数 （其值是一个数字，且内部以整数储存）， 时，返回 1 ，否则返回 0 。

lua_islightuserdata

[-0, +0, –]

```
int lua_islightuserdata (lua_State *L, int index);
```

当给定索引的值是一个轻量用户数据时，返回 1 ，否则返回 0 。

lua_isnil

[-0, +0, –]

```
int lua_isnil (lua_State *L, int index);
```

当给定索引的值是 **nil** 时，返回 1 ，否则返回 0 。

lua_isnone

[-0, +0, –]

```
int lua_isnone (lua_State *L, int index);
```

当给定索引无效时，返回 1 ，否则返回 0 。

lua_isnoneornil

[-0, +0, –]

```
int lua_isnoneornil (lua_State *L, int index);
```

当给定索引无效或其值是 **nil** 时， 返回 1 ，否则返回 0 。

lua_isnumber

[-0, +0, –]

```
int lua_isnumber (lua_State *L, int index);
```

当给定索引的值是一个数字，或是一个可转换为数字的字符串时，返回 1 ，否则返回 0 。

lua_isstring

[-0, +0, –]

```
int lua_isstring (lua_State *L, int index);
```

当给定索引的值是一个字符串或是一个数字 （数字总能转换成字符串）时，返回 1 ，否则返回 0 。

lua_istable

[-0, +0, –]

```
int lua_istable (lua_State *L, int index);
```

当给定索引的值是一张表时，返回 1 ，否则返回 0 。

lua_isthread

[-0, +0, –]

```
int lua_isthread (lua_State *L, int index);
```

当给定索引的值是一条线程时，返回 1 ，否则返回 0 。

`lua_isuserdata`

[-0, +0, –]

```
int lua_isuserdata (lua_State *L, int index);
```

当给定索引的值是一个用户数据（无论是完全的还是轻量的）时， 返回 1 ，否则返回 0 。

`lua_isyieldable`

[-0, +0, –]

```
int lua_isyieldable (lua_State *L);
```

如果给定的协程可以让出，返回 1 ，否则返回 0 。

`lua_KContext`

```
typedef ... lua_KContext;
```

延续函数上下文参数的类型。 这一定是一个数字类型。 当有 `intptr_t` 时，被定义为 `intptr_t` ， 因此它也可以保存指针。 否则，它被定义为 `ptrdiff_t` 。

`lua_KFunction`

```
typedef int (*lua_KFunction) (lua_State *L, int status, lua_KContext ctx);
```

延续函数的类型（参见 §4.7 ）。

`lua_len`

[-0, +1, *e*]

```
void lua_len (lua_State *L, int index);
```

返回给定索引的值的长度。 它等价于 Lua 中的 '`#`' 操作符 （参见 §3.4.7）。 它有可能触发 "length" 事件对应的元方法 （参见 §2.4 ）。 结果压栈。

`lua_load`

[-0, +1, –]

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname,
              const char *mode);
```

加载一段 Lua 代码块，但不运行它。 如果没有错误， `lua_load` 把一个编译好的代码块作为一个 Lua 函数压到栈顶。 否则， 压入错误消息。

`lua_load` 的返回值可以是：

- `LUA_OK`：没有错误；
- **`LUA_ERRSYNTAX`**：在预编译时碰到语法错误；
- `LUA_ERRMEM`：内存分配错误；
- `LUA_ERRGCMM`：在运行 `__gc` 元方法时出错了。 （这个错误和代码块加载过程无关，它是由垃圾收集器引发的。）

`lua_load` 函数使用一个用户提供的 `reader` 函数来读取代码块（参见 `lua_Reader` ）。 `data` 参数会被传入 `reader` 函数。

`chunkname` 这个参数可以赋予代码块一个名字， 这个名字被用于出错信息和调试信息（参见 §4.9）。

`lua_load` 会自动检测代码块是文本的还是二进制的， 然后做对应的加载操作（参见程序 `luac` ）。 字符串 `mode` 的作用和函数 `load` 一致。 它还可以是 `NULL` 等价于字符串 "`bt`"。

`lua_load` 的内部会使用栈， 因此 reader 函数必须永远在每次返回时保留栈的原样。

如果返回的函数有上值， 第一个上值会被设置为 保存在注册表（参见 §4.5） `LUA_RIDX_GLOBALS` 索引处的全局环境。 在加载主代码块时， 这个上值是 `_ENV` 变量（参见 §2.2）。 其它上值均被初始化为 **nil**。

`lua_newstate`

[-0, +0, –]

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

创建一个运行在新的独立的状态机中的线程。 如果无法创建线程或状态机（由于内存有限）则返回 `NULL` 。 参数 `f` 是一个分配器函数； Lua 将通过这个函数做状态机内所有的内存分配操作。 第二个参数 `ud` ， 这个指针将在每次调用分配器时被转入。

`lua_newtable`

[-0, +1, *e*]

```
void lua_newtable (lua_State *L);
```

创建一张空表，并将其压栈。 它等价于 `lua_createtable(L, 0, 0)` 。

`lua_newthread`

[-0, +1, *e*]

```
lua_State *lua_newthread (lua_State *L);
```

创建一条新线程，并将其压栈， 并返回维护这个线程的 `lua_State` 指针。 这个函数返回的新线程共享原线程的全局环境， 但是它有独立的运行栈。

没有显式的函数可以用来关闭或销毁掉一个线程。 线程跟其它 Lua 对象一样是垃圾收集的条目之一。

`lua_newuserdata`

[-0, +1, *e*]

```
void *lua_newuserdata (lua_State *L, size_t size);
```

这个函数分配一块指定大小的内存块， 把内存块地址作为一个完全用户数据压栈， 并返回这个地址。 宿主程序可以随意使用这块内存。

`lua_next`

[-1, +(2|0), *e*]

```
int lua_next (lua_State *L, int index);
```

从栈顶弹出一个键， 然后把索引指定的表中的一个键值对压栈 （弹出的键之后的 “下一” 对） 。 如果表中以无更多元素， 那么 `lua_next` 将返回 0 （什么也不压栈） 。

典型的遍历方法是这样的：

```
     /*  表放在索引 't' 处 */
     lua_pushnil(L);  /* 第一个键 */
     while (lua_next(L, t) != 0) {
       /* 使用 '键' （在索引 -2 处） 和 '值' （在索引 -1 处）*/
       printf("%s - %s\n",
              lua_typename(L, lua_type(L, -2)),
              lua_typename(L, lua_type(L, -1)));
       /* 移除 '值' ；保留 '键' 做下一次迭代 */
       lua_pop(L, 1);
     }
```

在遍历一张表的时候， 不要直接对键调用 `lua_tolstring` ， 除非你知道这个键一定是一个字符串。 调用 `lua_tolstring` 有可能改变给定索引位置的值； 这会对下一次调用 `lua_next` 造成影响。

关于迭代过程中修改被迭代的表的注意事项参见 `next` 函数。

lua_Number

```
typedef double lua_Number;
```

Lua 中浮点数的类型。

Lua 中数字的类型。缺省是 double ，但是你可以改成 float 。 （参见 `luaconf.h` 中的 `LUA_REAL` 。）

lua_numbertointeger

```
int lua_numbertointeger (lua_Number n, lua_Integer *p);
```

将一个 Lua 浮点数转换为一个 Lua 整数。 这个宏假设 `n` 有对应的整数值。 如果该值在 Lua 整数可表示范围内， 就将其转换为一个整数赋给 `*p` 。 宏的结果是一个布尔量，表示转换是否成功。 （注意、由于圆整关系，这个范围测试不用此宏很难做对。）

该宏有可能对其参数做多次取值。

lua_pcall

[-(nargs + 1), +(nresults|1), –]

```
int lua_pcall (lua_State *L, int nargs, int nresults, int msgh);
```

以保护模式调用一个函数。

`nargs` 和 `nresults` 的含义与 `lua_call` 中的相同。 如果在调用过程中没有发生错误， `lua_pcall` 的行为和 `lua_call` 完全一致。 但是，如果有错误发生的话， `lua_pcall` 会捕获它， 然后把唯一的值（错误消息）压栈，然后返回错误码。 同 `lua_call` 一样， `lua_pcall` 总是把函数本身和它的参数从栈上移除。

如果 `msgh` 是 0 ， 返回在栈顶的错误消息就和原始错误消息完全一致。 否则， `msgh` 就被当成是 错误处理函数 在栈上的索引位置。 （在当前的实现里，这个索引不能是伪索引。） 在发生运行时错误时， 这个函数会被调用而参数就是错误消息。 错误处理函数的返回值将被 `lua_pcall` 作为错误消息返回在堆栈上。

典型的用法中，错误处理函数被用来给错误消息加上更多的调试信息， 比如栈跟踪信息。 这些信息在 `lua_pcall` 返回后， 由于栈已经展开，所以收集不到了。

`lua_pcall` 函数会返回下列常数 （定义在 `lua.h` 内）中的一个：

- **`LUA_OK` (0):** 成功。
- **`LUA_ERRRUN`:** 运行时错误。
- **`LUA_ERRMEM`:** 内存分配错误。对于这种错，Lua 不会调用错误处理函数。
- **`LUA_ERRERR`:** 在运行错误处理函数时发生的错误。
- **`LUA_ERRGCMM`:** 在运行 `__gc` 元方法时发生的错误。 （这个错误和被调用的函数无关。）

`lua_pcallk`

[-(nargs + 1), +(nresults|1), –]

```
int lua_pcallk (lua_State *L,
                int nargs,
                int nresults,
                int msgh,
                lua_KContext ctx,
                lua_KFunction k);
```

这个函数的行为和 `lua_pcall` 完全一致，只不过它还允许被调用的函数让出 （参见 §4.7）。

`lua_pop`

[-n, +0, –]

```
void lua_pop (lua_State *L, int n);
```

从栈中弹出 `n` 个元素。

`lua_pushboolean`

[-0, +1, –]

```
void lua_pushboolean (lua_State *L, int b);
```

把 `b` 作为一个布尔量压栈。

`lua_pushcclosure`

[-n, +1, *e*]

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

把一个新的 C 闭包压栈。

当创建了一个 C 函数后，你可以给它关联一些值，这就是在创建一个 C 闭包（参见 §4.4）；接下来无论函数何时被调用，这些值都可以被这个函数访问到。为了将一些值关联到一个 C 函数上，首先这些值需要先被压入堆栈（如果有多个值，第一个先压）。接下来调用 `lua_pushcclosure` 来创建出闭包并把这个 C 函数压到栈上。参数 `n` 告之函数有多少个值需要关联到函数上。`lua_pushcclosure` 也会把这些值从栈上弹出。

`n` 的最大值是 255 。

当 `n` 为零时，这个函数将创建出一个轻量 *C* 函数，它就是一个指向 C 函数的指针。这种情况下，不可能抛出内存错误。

`lua_pushcfunction`

[-0, +1, –]

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

将一个 C 函数压栈。这个函数接收一个 C 函数指针，并将一个类型为 `function` 的 Lua 值压栈。当这个栈顶的值被调用时，将触发对应的 C 函数。

注册到 Lua 中的任何函数都必须遵循正确的协议来接收参数和返回值（参见 `lua_CFunction` ）。

`lua_pushcfunction` 是作为一个宏定义出现的：

```
     #define lua_pushcfunction(L,f)  lua_pushcclosure(L,f,0)
```

`lua_pushfstring`

[-0, +1, *e*]

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

把一个格式化过的字符串压栈，然后返回这个字符串的指针。它和 C 函数 `sprintf` 比较像，不过有一些重要的区别：

- 你不需要为结果分配空间：其结果是一个 Lua 字符串，由 Lua 来关心其内存分配（同时通过垃圾收集来释放内存）。
- 这个转换非常的受限。不支持符号、宽度、精度。转换符只支持 '`%%`' （插入一个字符

'`%`'），'`%s`'（插入一个带零终止符的字符串，没有长度限制），'`%f`'（插入一个 `lua_Number`），'`%L`'（插入一个 `lua_Integer`），'`%p`'（插入一个指针或是一个十六进制数），'`%d`'（插入一个 `int`），'`%c`'（插入一个用 `int` 表示的单字节字符），以及'`%U`'（插入一个用 `long int` 表示的 UTF-8 字）。

`lua_pushglobaltable`

[-0, +1, –]

```
void lua_pushglobaltable (lua_State *L);
```

将全局环境压栈。

`lua_pushinteger`

[-0, +1, –]

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

把值为 `n` 的整数压栈。

`lua_pushlightuserdata`

[-0, +1, –]

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

把一个轻量用户数据压栈。

用户数据是保留在 Lua 中的 C 值。 轻量用户数据 表示一个指针 `void*`。 它是一个像数字一样的值： 你不需要专门创建它，它也没有独立的元表，而且也不会被收集（因为从来不需要创建）。 只要表示的 C 地址相同，两个轻量用户数据就相等。

`lua_pushliteral`

[-0, +1, *e*]

```
const char *lua_pushliteral (lua_State *L, const char *s);
```

这个宏等价于 `lua_pushlstring`， 区别仅在于只能在 `s` 是一个字面量时才能用它。 它会自动给出字符串的长度。

`lua_pushlstring`

[-0, +1, *e*]

```
const char *lua_pushlstring (lua_State *L, const char *s, size_t len);
```

把指针 `s` 指向的长度为 `len` 的字符串压栈。 Lua 对这个字符串做一个内部副本（或是复用一个副本）， 因此 `s` 处的内存在函数返回后，可以释放掉或是立刻重用于其它用途。 字符串内可以是任意二进制数据，包括零字符。

返回内部副本的指针。

`lua_pushnil`

[-0, +1, –]

```
void lua_pushnil (lua_State *L);
```

将空值压栈。

`lua_pushnumber`

[-0, +1, –]

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

把一个值为 `n` 的浮点数压栈。

`lua_pushstring`

[-0, +1, *e*]

```
const char *lua_pushstring (lua_State *L, const char *s);
```

将指针 s 指向的零结尾的字符串压栈。 因此 `s` 处的内存在函数返回后，可以释放掉或是立刻重用于其它用途。

返回内部副本的指针。

如果 `s` 为 `NULL`，将 **nil** 压栈并返回 `NULL` 。

`lua_pushthread`

[-0, +1, –]

```
int lua_pushthread (lua_State *L);
```

把 `L` 表示的线程压栈。 如果这个线程是当前状态机的主线程的话，返回 1 。

lua_pushvalue

[-0, +1, –]

```
void lua_pushvalue (lua_State *L, int index);
```

把栈上给定索引处的元素作一个副本压栈。

lua_pushvfstring

[-0, +1, *e*]

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```

等价于 `lua_pushfstring` ， 不过是用 `va_list` 接收参数，而不是用可变数量的实际参数。

lua_rawequal

[-0, +0, –]

```
int lua_rawequal (lua_State *L, int index1, int index2);
```

如果索引 `index1` 与索引 `index2` 处的值 本身相等（即不调用元方法）， 返回 1 。 否则返回 0 。 当任何一个索引无效时，也返回 0 。

lua_rawget

[-1, +1, –]

```
int lua_rawget (lua_State *L, int index);
```

类似于 `lua_gettable` ， 但是作一次直接访问（不触发元方法）。

lua_rawgeti

[-0, +1, –]

```
int lua_rawgeti (lua_State *L, int index, lua_Integer n);
```

把 `t[n]` 的值压栈， 这里的 `t` 是指给定索引处的表。 这是一次直接访问；就是说，它不会触发元方法。

返回入栈值的类型。

`lua_rawgetp`

[-0, +1, –]

```
int lua_rawgetp (lua_State *L, int index, const void *p);
```

把 `t[k]` 的值压栈， 这里的 `t` 是指给定索引处的表， `k` 是指针 `p` 对应的轻量用户数据。 这是一次直接访问；就是说，它不会触发元方法。

返回入栈值的类型。

`lua_rawlen`

[-0, +0, –]

```
size_t lua_rawlen (lua_State *L, int index);
```

返回给定索引处值的固有"长度"： 对于字符串，它指字符串的长度； 对于表；它指不触发元方法的情况下取长度操作（'`#`'）应得到的值； 对于用户数据，它指为该用户数据分配的内存块的大小； 对于其它值，它为 0 。

`lua_rawset`

[-2, +0, *e*]

```
void lua_rawset (lua_State *L, int index);
```

类似于 `lua_settable` ， 但是是做一次直接赋值（不触发元方法）。

`lua_rawseti`

[-1, +0, *e*]

```
void lua_rawseti (lua_State *L, int index, lua_Integer i);
```

等价于 `t[i] = v` ， 这里的 `t` 是指给定索引处的表， 而 `v` 是栈顶的值。

这个函数会将值弹出栈。 赋值是直接的；即不会触发元方法。

`lua_rawsetp`

[-1, +0, *e*]

```
void lua_rawsetp (lua_State *L, int index, const void *p);
```

等价于 `t[k] = v` ， 这里的 `t` 是指给定索引处的表， `k` 是指针 `p` 对应的轻量用户数据。 而 `v` 是栈顶的值。

这个函数会将值弹出栈。 赋值是直接的；即不会触发元方法。

`lua_Reader`

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

`lua_load` 用到的读取器函数， 每次它需要一块新的代码块的时候， `lua_load` 就调用读取器， 每次都会传入一个参数 `data` 。 读取器需要返回含有新的代码块的一块内存的指针， 并把 `size` 设为这块内存的大小。 内存块必须在下一次函数被调用之前一直存在。 读取器可以通过返回 `NULL` 或设 `size` 为 0 来指示代码块结束。 读取器可能返回多个块，每个块可以有任意的大于零的尺寸。

`lua_register`

[-0, +0, *e*]

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```

把 C 函数 `f` 设到全局变量 `name` 中。 它通过一个宏定义：

```
     #define lua_register(L,n,f) \
            (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

`lua_remove`

[-1, +0, –]

```
void lua_remove (lua_State *L, int index);
```

从给定有效索引处移除一个元素， 把这个索引之上的所有元素移下来填补上这个空隙。 不能用伪索引来调用这个函数，因为伪索引并不指向真实的栈上的位置。

`lua_replace`

[-1, +0, –]

```
void lua_replace (lua_State *L, int index);
```

把栈顶元素放置到给定位置而不移动其它元素 （因此覆盖了那个位置处的值）， 然后将栈顶元素弹出。

`lua_resume`

[-?, +?, –]

```
int lua_resume (lua_State *L, lua_State *from, int nargs);
```

在给定线程中启动或延续一条协程 。

要启动一个协程的话， 你需要把主函数以及它需要的参数压入线程栈； 然后调用 `lua_resume` ， 把 `nargs` 设为参数的个数。 这次调用会在协程挂起时或是结束运行后返回。 当函数返回时，堆栈中会有传给 `lua_yield` 的所有值， 或是主函数的所有返回值。 当协程让出， `lua_resume` 返回 `LUA_YIELD` ， 若协程结束运行且没有任何错误时，返回 0 。 如果有错则返回错误代码（参见 `lua_pcall` ）。

在发生错误的情况下， 堆栈没有展开， 因此你可以使用调试 API 来处理它。 错误消息放在栈顶在。

要延续一个协程， 你需要清除上次 `lua_yield` 遗留下的所有结果， 你把需要传给 `yield` 作结果的值压栈， 然后调用 `lua_resume` 。

参数 `from` 表示协程从哪个协程中来延续 `L` 的。 如果不存在这样一个协程，这个参数可以是 `NULL` 。

`lua_rotate`

[-0, +0, –]

```
void lua_rotate (lua_State *L, int idx, int n);
```

把从 `idx` 开始到栈顶的元素轮转 `n` 个位置。 对于 `n` 为正数时，轮转方向是向栈顶的； 当 `n` 为负数时，向栈底方向轮转 `-n` 个位置。 `n` 的绝对值不可以比参于轮转的切片长度大。

lua_setallocf

[-0, +0, –]

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

把指定状态机的分配器函数换成带上用户数据 `ud` 的 `f` 。

lua_setfield

[-1, +0, *e*]

```
void lua_setfield (lua_State *L, int index, const char *k);
```

做一个等价于 `t[k] = v` 的操作， 这里 `t` 是给出的索引处的值， 而 `v` 是栈顶的那个值。

这个函数将把这个值弹出栈。 跟在 Lua 中一样，这个函数可能触发一个 "newindex" 事件的元方法 （参见 §2.4）。

lua_setglobal

[-1, +0, *e*]

```
void lua_setglobal (lua_State *L, const char *name);
```

从堆栈上弹出一个值，并将其设为全局变量 `name` 的新值。

lua_seti

[-1, +0, *e*]

```
void lua_seti (lua_State *L, int index, lua_Integer n);
```

做一个等价于 `t[n] = v` 的操作， 这里 `t` 是给出的索引处的值， 而 `v` 是栈顶的那个值。

这个函数将把这个值弹出栈。 跟在 Lua 中一样，这个函数可能触发一个 "newindex" 事件的元方法 （参见 §2.4）。

lua_setmetatable

[-1, +0, –]

```
void lua_setmetatable (lua_State *L, int index);
```

把一张表弹出栈，并将其设为给定索引处的值的元表。

lua_settable

[-2, +0, *e*]

```
void lua_settable (lua_State *L, int index);
```

做一个等价于 `t[k] = v` 的操作， 这里 `t` 是给出的索引处的值， `v` 是栈顶的那个值， `k` 是栈顶之下的值。

这个函数会将键和值都弹出栈。 跟在 Lua 中一样，这个函数可能触发一个 "newindex" 事件的元方法 （参见 §2.4）。

lua_settop

[-?, +?, –]

```
void lua_settop (lua_State *L, int index);
```

参数允许传入任何索引以及 0 。 它将把堆栈的栈顶设为这个索引。 如果新的栈顶比原来的大， 超出部分的新元素将被填为 **nil** 。 如果 `index` 为 0 ， 把栈上所有元素移除。

lua_setuservalue

[-1, +0, –]

```
void lua_setuservalue (lua_State *L, int index);
```

从栈上弹出一个值并将其设为给定索引处用户数据的关联值。

lua_State

```
typedef struct lua_State lua_State;
```

一个不透明的结构， 它指向一条线程并间接（通过该线程）引用了整个 Lua 解释器的状态。 Lua 库是完全可重入的： 它没有任何全局变量。 状态机所有的信息都可以通过这个结构访问到。

这个结构的指针必须作为第一个参数传递给每一个库函数。 `lua_newstate` 是一个例外， 这个函数会从头创建一个 Lua 状态机。

lua_status

[-0, +0, –]

```
int lua_status (lua_State *L);
```

返回线程 `L` 的状态。

正常的线程状态是 0 （ `LUA_OK` ）。 当线程用 `lua_resume` 执行完毕并抛出了一个错误时，状态值是错误码。 如果线程被挂起，状态为 `LUA_YIELD` 。

你只能在状态为 `LUA_OK` 的线程中调用函数。 你可以延续一个状态为 `LUA_OK` 的线程 （用于开始新协程）或是状态为 `LUA_YIELD` 的线程 （用于延续协程）。

lua_stringtonumber

[-0, +1, –]

```
size_t lua_stringtonumber (lua_State *L, const char *s);
```

将一个零结尾的字符串 `s` 转换为一个数字， 将这个数字压栈，并返回字符串的总长度（即长度加一）。 转换的结果可能是整数也可能是浮点数， 这取决于 Lua 的转换语法（参见 §3.1）。 这个字符串可以有前置和后置的空格以及符号。 如果字符串并非一个有效的数字，返回 0 并不把任何东西压栈。 （注意，这个结果可以当成一个布尔量使用，为真即转换成功。）

lua_toboolean

[-0, +0, –]

```
int lua_toboolean (lua_State *L, int index);
```

把给定索引处的 Lua 值转换为一个 C 中的布尔量（ 0 或是 1 ）。 和 Lua 中做的所有测试一样， `lua_toboolean` 会把任何不同于 **false** 和 **nil** 的值当作真返回； 否则就返回假。 （如果你想只接收真正的 boolean 值， 就需要使用 `lua_isboolean` 来测试值的类型。）

lua_tocfunction

[-0, +0, –]

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

把给定索引处的 Lua 值转换为一个 C 函数。 这个值必须是一个 C 函数； 如果不是就返回 `NULL` 。

`lua_tointeger`

[-0, +0, –]

```
lua_Integer lua_tointeger (lua_State *L, int index);
```

等价于调用 `lua_tointegerx`， 其参数 `isnum` 为 `NULL` 。

`lua_tointegerx`

[-0, +0, –]

```
lua_Integer lua_tointegerx (lua_State *L, int index, int *isnum);
```

将给定索引处的 Lua 值转换为带符号的整数类型 `lua_Integer` 。 这个 Lua 值必须是一个整数，或是一个可以被转换为整数 （参见 §3.4.3）的数字或字符串； 否则， `lua_tointegerx` 返回 0 。

如果 `isnum` 不是 `NULL` ， `*isnum` 会被设为操作是否成功。

`lua_tolstring`

[-0, +0, *e*]

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

把给定索引处的 Lua 值转换为一个 C 字符串。 如果 `len` 不为 `NULL` ， 它还把字符串长度设到 `*len` 中。 这个 Lua 值必须是一个字符串或是一个数字； 否则返回返回 `NULL` 。 如果值是一个数字， `lua_tolstring` 还会 把堆栈中的那个值的实际类型转换为一个字符串。 （当遍历一张表的时候， 若把 `lua_tolstring` 作用在键上， 这个转换有可能导致 `lua_next` 弄错。）

`lua_tolstring` 返回一个已对齐指针 指向 Lua 状态机中的字符串。 这个字符串总能保证 （ C 要求的）最后一个字符为零 ('\0') ， 而且它允许在字符串内包含多个这样的零。

因为 Lua 中可能发生垃圾收集， 所以不保证 `lua_tolstring` 返回的指针， 在对应的值从堆栈中移除后依然有效。

`lua_tonumber`

[-0, +0, –]

```
lua_Number lua_tonumber (lua_State *L, int index);
```

等价于调用 `lua_tonumberx`， 其参数 `isnum` 为 `NULL`。

`lua_tonumberx`

[-0, +0, –]

```
lua_Number lua_tonumberx (lua_State *L, int index, int *isnum);
```

把给定索引处的 Lua 值转换为 `lua_Number` 这样一个 C 类型 （参见 lua_Number ）。 这个 Lua 值必须是一个数字或是一个可转换为数字的字符串 （参见 §3.4.3）； 否则， `lua_tonumberx` 返回 0 。

如果 `isnum` 不是 `NULL`， `*isnum` 会被设为操作是否成功。

`lua_topointer`

[-0, +0, –]

```
const void *lua_topointer (lua_State *L, int index);
```

把给定索引处的值转换为一般的 C 指针 (`void*`) 。 这个值可以是一个用户对象，表 ，线程或是一个函数； 否则， `lua_topointer` 返回 `NULL` 。 不同的对象有不同的指针。 不存在把指针再转回原有类型的方法。

这个函数通常只用于调试信息。

`lua_tostring`

[-0, +0, *e*]

```
const char *lua_tostring (lua_State *L, int index);
```

等价于调用 `lua_tolstring` ， 其参数 `len` 为 `NULL` 。

`lua_tothread`

[-0, +0, –]

```
lua_State *lua_tothread (lua_State *L, int index);
```

把给定索引处的值转换为一个 Lua 线程 （表示为 `lua_State*` ）。 这个值必须是一个线程 ；否则函数返回 `NULL` 。

`lua_touserdata`

[-0, +0, –]

```
void *lua_touserdata (lua_State *L, int index);
```

如果给定索引处的值是一个完全用户数据， 函数返回其内存块的地址。 如果值是一个轻量用户数据， 那么就返回它表示的指针。 否则，返回 `NULL` 。

`lua_type`

[-0, +0, –]

```
int lua_type (lua_State *L, int index);
```

返回给定有效索引处值的类型， 当索引无效（或无法访问）时则返回 `LUA_TNONE` 。 `lua_type` 返回的类型被编码为一些个在 `lua.h` 中定义的常量： `LUA_TNIL` , `LUA_TNUMBER` , `LUA_TBOOLEAN` , `LUA_TSTRING` , `LUA_TTABLE` , `LUA_TFUNCTION` , `LUA_TUSERDATA` , `LUA_TTHREAD` , `LUA_TLIGHTUSERDATA` 。

`lua_typename`

[-0, +0, –]

```
const char *lua_typename (lua_State *L, int tp);
```

返回 `tp` 表示的类型名， 这个 `tp` 必须是 `lua_type` 可能返回的值中之一。

`lua_Unsigned`

```
typedef ... lua_Unsigned;
```

`lua_Integer` 的无符号版本。

`lua_upvalueindex`

[-0, +0, –]

```
int lua_upvalueindex (int i);
```

返回当前运行的函数（参见 §4.4）的第 `i` 个上值的伪索引。

`lua_version`

[-0, +0, *v*]

```
const lua_Number *lua_version (lua_State *L);
```

返回保存在 Lua 内核中储存的版本数字的地址。 当调用时传入一个合法的 `lua_State` ， 返回创建该状态机时的版本地址。 如果用 `NULL` 调用， 返回调用者的版本地址。

`lua_Writer`

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

被 `lua_dump` 用到的写入器函数。 每次 `lua_dump` 产生了一段新的代码块， 它都会调用写入器。 传入要写入的缓冲区 (`p`) 和它的尺寸 (`sz`) ， 以及传给 `lua_dump` 的参数 `data` 。

写入器会返回一个错误码： 0 表示没有错误； 别的值均表示一个错误， 并且会让 `lua_dump` 停止再次调用写入器。

`lua_xmove`

[-?, +?, –]

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

交换同一个状态机下不同线程中的值。

这个函数会从 `from` 的栈上弹出 `n` 个值， 然后把它们压入 `to` 的栈上。

`lua_yield`

[-?, +?, *e*]

```
int lua_yield (lua_State *L, int nresults);
```

这个函数等价于调用 `lua_yieldk`， 不同的是不提供延续函数（参见 §4.7）。 因此，当线程被延续，线程会继续运行调用 `lua_yield` 函数的函数。

`lua_yieldk`

[-?, +?, *e*]

```
int lua_yieldk (lua_State *L,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```

让出协程（线程）。

当 C 函数调用了 `lua_yieldk`， 当前运行的协程会挂起， 启动这个线程的 `lua_resume` 调用返回。 参数 `nresults` 指栈上需返回给 `lua_resume` 的返回值的个数。

当协程再次被延续时， Lua 调用延续函数 `k` 继续运行被挂起（参见 §4.7）的 C 函数。 延续函数会从前一个函数中接收到相同的栈， 栈中的 `n` 个返回值被移除而压入了从 `lua_resume` 传入的参数。 此外，延续函数还会收到传给 `lua_yieldk` 的参数 `ctx`。

通常，这个函数不会返回； 当协程一次次延续，将从延续函数继续运行。 然而，有一个例外： 当这个函数从一个逐行运行的钩子函数（参见 §4.9） 中调用时， `lua_yieldk` 不可以提供延续函数。 （也就是类似 `lua_yield` 的形式）， 而此时，钩子函数在调用完让出后将立刻返回。 Lua 会使协程让出，一旦协程再次被延续， 触发钩子的函数会继续正常运行。

当一个线程处于未提供延续函数的 C 调用中，调用它会抛出一个错误。 从并非用延续方式（例如：主线程）启动的线程中调用它也会这样。

4.9 – 调试接口

Lua 没有内置的调试机制。 但是它提供了一组特殊的函数接口以及 钩子。 这组接口可用于构建出不同的调试器、性能剖析器、 或是其它需要从解释器获取“内部信息”的工具。

`lua_Debug`

```
typedef struct lua_Debug {
  int event;
  const char *name;           /* (n) */
  const char *namewhat;       /* (n) */
  const char *what;           /* (S) */
  const char *source;         /* (S) */
  int currentline;            /* (l) */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  unsigned char nups;         /* (u) 上值的数量 */
  unsigned char nparams;      /* (u) 参数的数量 */
  char isvararg;              /* (u) */
  char istailcall;            /* (t) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* 私有部分 */
  _其它域_
} lua_Debug;
```

这是一个携带有有关函数或活动记录的各种信息的结构。 `lua_getstack` 只会填充结构的私有部分供后面使用。 调用 `lua_getinfo` 可以在 `lua_Debug` 中填充那些可被使用的信息域。

下面对 `lua_Debug` 的各个域做一个说明：

- **`source`**：创建这个函数的代码块的名字。 如果 `source` 以 '`@`' 打头， 指这个函数定义在一个文件中，而 '`@`' 之后的部分就是文件名。 若 `source` 以 '`=`' 打头， 剩余的部分由用户行为来决定如何表示源码。 其它的情况下，这个函数定义在一个字符串中， 而 `source` 正是那个字符串。
- **`short_src`**：一个“可打印版本”的 `source` ， 用于出错信息。
- **`linedefined`**：函数定义开始处的行号。
- **`lastlinedefined`**：函数定义结束处的行号。
- **`what`**：如果函数是一个 Lua 函数，则为一个字符串 `"Lua"` ； 如果是一个 C 函数，则为 `"C"` ； 如果它是一个代码块的主体部分，则为 `"main"` 。
- **`currentline`**：给定函数正在执行的那一行。 当提供不了行号信息的时候， `currentline` 被设为 -1 。
- **`name`**：给定函数的一个合理的名字。 因为 Lua 中的函数是一等公民， 所以它们没有固定的名字： 一些函数可能是全局复合变量的值， 另一些可能仅仅只是被保存在一张表的某个域中。 `lua_getinfo` 函数会检查函数是怎样被调用的， 以此来找到一个适合的名字。 如果它找不到名字， `name` 就被设置为 `NULL` 。
- **`namewhat`**：用于解释 `name` 域。 `namewhat` 的值可以是 `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"`, 或是 `""` （空串）。 这取决于函数怎样被调用。 （Lua 用空串表示其它选项都不符合。）
- **`istailcall`**：如果函数以尾调用形式调用，这个值就为真。 在这种情况下，当层的调用者不在栈中。
- **`nups`**：函数的上值个数。
- **`nparams`**：函数固定形参个数 （对于 C 函数永远是 0 ）。
- **`isvararg`**：如果函数是一个可变参数函数则为真 （对于 C 函数永远为真）。

`lua_gethook`

[-0, +0, –]

```
lua_Hook lua_gethook (lua_State *L);
```

返回当前的钩子函数。

lua_gethookcount

[-0, +0, –]

```
int lua_gethookcount (lua_State *L);
```

返回当前的钩子计数。

lua_gethookmask

[-0, +0, –]

```
int lua_gethookmask (lua_State *L);
```

返回当前的钩子掩码。

lua_getinfo

[-(0|1), +(0|1|2), *e*]

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

返回一个指定的函数或函数调用的信息。

当用于取得一次函数调用的信息时， 参数 `ar` 必须是一个有效的活动的记录。 这条记录可以是前一次调用 `lua_getstack` 得到的， 或是一个钩子 （参见 `lua_Hook` ）得到的参数。

用于获取一个函数的信息时， 可以把这个函数压入堆栈， 然后把 `what` 字符串以字符 '>' 起头。 （这会让 `lua_getinfo` 从栈顶上弹出函数。） 例如，想知道函数 `f` 是在哪一行定义的， 你可以使用下列代码：

```
     lua_Debug ar;
     lua_getglobal(L, "f");  /* 取得全局变量 'f' */
     lua_getinfo(L, ">S", &ar);
     printf("%d\n", ar.linedefined);
```

`what` 字符串中的每个字符都筛选出结构 `ar` 结构中一些域用于填充， 或是把一个值压入堆栈：

- '`n`': 填充 `name` 及 `namewhat` 域；
- '`S`': 填充 `source` ， `short_src` ， `linedefined` ， `lastlinedefined` ，以及 `what` 域；
- '`l`': 填充 `currentline` 域；
- '`t`': 填充 `istailcall` 域；
- '`u`': 填充 `nups` ， `nparams` ，及 `isvararg` 域；
- '`f`': 把正在运行中指定层次处函数压栈；
- '`L`': 将一张表压栈，这张表中的整数索引用于描述函数中哪些行是有效行。（有效行指有实际代码的行，即你可以置入断点的行。无效行包括空行和只有注释的行。）

  如果这个选项和选项 '`f`' 同时使用，这张表在函数之后压栈。

这个函数出错会返回 0 （例如， `what` 中有一个无效选项）。

`lua_getlocal`

[-0, +(0|1), –]

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
```

从给定活动记录或从一个函数中获取一个局部变量的信息。

对于第一种情况，参数 `ar` 必须是一个有效的活动的记录。这条记录可以是前一次调用 `lua_getstack` 得到的，或是一个钩子（参见 `lua_Hook` ）的参数。索引 `n` 用于选择要检阅哪个局部变量；参见 `debug.getlocal` 中关于变量的索引和名字的介绍。

`lua_getlocal` 将变量的值压栈，并返回其名字。

对于第二种情况， `ar` 必须填 `NULL` 。需要探知的函数必须放在栈顶。对于这种情况，只有 Lua 函数的形参是可见的（没有关于还有哪些活动变量的信息）也不会有任何值压栈。

当索引大于活动的局部变量的数量，返回 `NULL` （无任何压栈）

`lua_getstack`

[-0, +0, –]

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

获取解释器的运行时栈的信息。

这个函数用正在运行中的指定层次处函数的 活动记录 来填写 `lua_Debug` 结构的一部分。 0 层表示当前运行的函数， *n+1* 层的函数就是调用第 *n* 层（尾调用例外，它不算在栈层次中）函数的那一个。如果没有错误， `lua_getstack` 返回 1 ；当调用传入的层次大于堆栈深度的

时候，返回 0 。

lua_getupvalue

[-0, +(0|1), –]

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

获取一个闭包的上值信息。 （对于 Lua 函数，上值是函数需要使用的外部局部变量， 因此这些变量被包含在闭包中。） `lua_getupvalue` 获取第 `n` 个上值， 把这个上值的值压栈， 并且返回它的名字。 `funcindex` 指向闭包在栈上的位置。 （ 因为上值在整个函数中都有效，所以它们没有特别的次序。 因此，它们以字母次序来编号。）

当索引号比上值数量大的时候， 返回 `NULL` （而且不会压入任何东西）。 对于 C 函数，所有上值的名字都是空串 `""` 。

lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

用于调试的钩子函数类型。

无论何时钩子被调用，它的参数 `ar` 中的 `event` 域都被设为触发钩子的事件。 Lua 把这些事件定义为以下常量： `LUA_HOOKCALL` ， `LUA_HOOKRET` ， `LUA_HOOKTAILCALL` ， `LUA_HOOKLINE` ， `LUA_HOOKCOUNT` 。 除此之外，对于 line 事件， `currentline` 域也被设置。 要想获得 `ar` 中的其他域， 钩子必须调用 `lua_getinfo` 。

对于 call 事件， `event` 可以是 `LUA_HOOKCALL` 这个通常值， 或是 `LUA_HOOKTAILCALL` 表示尾调用； 后一种情况，没有对应的返回事件。

当 Lua 运行在一个钩子内部时， 它将屏蔽掉其它对钩子的调用。 也就是说，如果一个钩子函数内再调回 Lua 来执行一个函数或是一个代码块 ， 这个执行操作不会触发任何的钩子。

钩子函数不能有延续点， 即不能用一个非空的 `k` 调用 `lua_yieldk` ， `lua_pcallk` ，或 `lua_callk` 。

钩子函数可以在满足下列条件时让出： 只有行计数事件可以让出，且不能在让出时传出任何值； 从钩子里让出必须用 `lua_yield` 来结束钩子的运行，且 `nresults` 必须为零。

lua_sethook

[-0, +0, –]

```
void lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```

设置一个调试用钩子函数。

参数 `f` 是钩子函数。 `mask` 指定在哪些事件时会调用： 它由下列一组位常量构成 `LUA_MASKCALL` ， `LUA_MASKRET` ， `LUA_MASKLINE` ， `LUA_MASKCOUNT` 。 参数 `count` 只在掩码中包含有 `LUA_MASKCOUNT` 才有意义。 对于每个事件，钩子被调用的情况解释如下：

- **call hook:** 在解释器调用一个函数时被调用。 钩子将于 Lua 进入一个新函数后， 函数获取参数前被调用。
- **return hook:** 在解释器从一个函数中返回时调用。 钩子将于 Lua 离开函数之前的那一刻被调用。 没有标准方法来访问被函数返回的那些值。
- **line hook:** 在解释器准备开始执行新的一行代码时， 或是跳转到这行代码中时（即使在同一行内跳转）被调用。 （这个事件仅仅在 Lua 执行一个 Lua 函数时发生。）
- **count hook:** 在解释器每执行 `count` 条指令后被调用。 （这个事件仅仅在 Lua 执行一个 Lua 函数时发生。）

钩子可以通过设置 `mask` 为零屏蔽。

`lua_setlocal`

[-(0|1), +0, –]

```
const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
```

设置给定活动记录中的局部变量的值。 参数 `ar` 与 `n` 和 `lua_getlocal` 中的一样 （参见 `lua_getlocal` ）。 `lua_setlocal` 把栈顶的值赋给变量然后返回变量的名字。 它会将值从栈顶弹出。

当索引大于活动局部变量的数量时，返回 `NULL` （什么也不弹出）。

`lua_setupvalue`

[-(0|1), +0, –]

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

设置闭包上值的值。 它把栈顶的值弹出并赋于上值并返回上值的名字。 参数 `funcindex` 与 `n` 和 `lua_getupvalue` 中的一样 （参见 `lua_getupvalue` ）。

当索引大于上值的数量时，返回 `NULL` （什么也不弹出）。

`lua_upvalueid`

[-0, +0, –]

```
void *lua_upvalueid (lua_State *L, int funcindex, int n);
```

返回索引 `funcindex` 处的闭包中 编号为 `n` 的上值的一个唯一标识符。 参数 `funcindex` 与 `n` 和 `lua_getupvalue` 中的一样 （参见 `lua_getupvalue` ）。 （但 `n` 不可以大于上值的数量）。

这些唯一标识符可用于检测不同的闭包是否共享了相同的上值。 共享同一个上值的 Lua 闭包（即它们指的同一个外部局部变量） 会针对这个上值返回相同的标识。

`lua_upvaluejoin`

[-0, +0, –]

```
void lua_upvaluejoin (lua_State *L, int funcindex1, int n1,
                                    int funcindex2, int n2);
```

让索引 `funcindex1` 处的 Lua 闭包的第 `n1` 个上值 引用索引 `funcindex2` 处的 Lua 闭包的第 `n2` 个上值。

5 – 辅助库

辅助库 提供了一些便捷函数，方便在 C 中为 Lua 编程。 基础 API 提供了 C 和 Lua 交互用的主要函数， 而辅助库则为一些常见的任务提供了高阶函数。

所有辅助库中的函数和类型都定义在头文件 `lauxlib.h` 中， 它们均带有前缀 `luaL_` 。

辅助库中的所有函数都基于基础 API 实现。 故而它们并没有提供任何基础 API 实现不了的功能。 虽然如此，使用辅助库可以让你的代码更为健壮。

一些辅助库函数会在内部使用一些额外的栈空间。 当辅助库使用的栈空间少于五个时， 它们不去检查栈大小；而是简单的假设栈够用。

一些辅助库中的函数用于检查 C 函数的参数。 因为错误信息格式化为指代参数 （例如，" `bad argument #1` "）， 你就不要把这些函数用于参数之外的值了。

如果检查无法通过， `luaL_check*` 这些函数一定会抛出错误。

5.1 – 函数和类型

这里我们按字母表次序列出了辅助库中的所有函数和类型。

`luaL_addchar`

[-?, +?, *e*]

```
void luaL_addchar (luaL_Buffer *B, char c);
```

向缓存 `B` （参见 `luaL_Buffer` ） 添加一个字节 `c` 。

`luaL_addlstring`

[-?, +?, *e*]

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

向缓存 `B` （参见 `luaL_Buffer` ） 添加一个长度为 `l` 的字符串 `s` 。 这个字符串可以包含零。

`luaL_addsize`

[-?, +?, *e*]

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

向缓存 `B` （参见 `luaL_Buffer` ） 添加一个已在之前复制到缓冲区（参见 `luaL_prepbuffer` ） 的长度为 `n` 的字符串。

`luaL_addstring`

[-?, +?, *e*]

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

向缓存 `B` （参见 `luaL_Buffer` ） 添加一个零结尾的字符串 `s` 。

`luaL_addvalue`

[-1, +?, *e*]

```
void luaL_addvalue (luaL_Buffer *B);
```

向缓存 `B` （参见 `luaL_Buffer` ） 添加栈顶的一个值，随后将其弹出。

这个函数是操作字符串缓存的函数中，唯一一个会（且必须）在栈上放置额外元素的。 这个元素将被加入缓存。

`luaL_argcheck`

[-0, +0, *v*]

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int arg,
                    const char *extramsg);
```

检查 `cond` 是否为真。 如果不为真，以标准信息形式抛出一个错误 （参见 `luaL_argerror` ） 。

`luaL_argerror`

[-0, +0, *v*]

```
int luaL_argerror (lua_State *L, int arg, const char *extramsg);
```

抛出一个错误报告调用的 C 函数的第 `arg` 个参数的问题。 它使用下列标准信息并包含了一段 `extramsg` 作为注解：

```
     bad argument #_arg_ to '_funcname_' (_extramsg_)
```

这个函数永远不会返回。

`luaL_Buffer`

```
typedef struct luaL_Buffer luaL_Buffer;
```

字符串缓存 的类型。

字符串缓存可以让 C 代码分段构造一个 Lua 字符串。 使用模式如下：

- 首先定义一个类型为 `luaL_Buffer` 的变量 `b` 。
- 调用 `luaL_buffinit(L, &b)` 初始化它。
- 然后调用 `luaL_add*` 这组函数向其添加字符串片断。
- 最后调用 `luaL_pushresult(&b)` 。 最后这次调用会在栈顶留下最终的字符串。

如果你预先知道结果串的长度， 你可以这样使用缓存：

- 首先定义一个类型为 `luaL_Buffer` 的变量 `b` 。
- 然后调用 `luaL_buffinitsize(L, &b, sz)` 预分配 `sz` 大小的空间。
- 接着将字符串复制入这个空间。
- 最后调用 `luaL_pushresultsize(&b, sz)` ， 这里的 `sz` 指已经复制到缓存内的字符串长度。

一般的操作过程中，字符串缓存会使用不定量的栈槽。 因此，在使用缓存中，你不能假定目前栈顶在哪。 在对缓存操作的函数调用间，你都可以使用栈，只需要保证栈平衡即可； 即，在你做一次缓存操作调用时，当时的栈位置和上次调用缓存操作后的位置相同。 （对于 `luaL_addvalue` 是个唯一的例外。） 在调用完 `luaL_pushresult` 后， 栈会恢复到缓存初始化时的位置上，并在顶部压入最终的字符串。

`luaL_buffinit`

[-0, +0, –]

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

初始化缓存 `B` 。 这个函数不会分配任何空间； 缓存必须以一个变量的形式声明 （参见 `luaL_Buffer` ） 。

`luaL_buffinitsize`

[-?, +?, *e*]

```
char *luaL_buffinitsize (lua_State *L, luaL_Buffer *B, size_t sz);
```

等价于调用序列 `luaL_buffinit` ， `luaL_prepbuffsize` 。

`luaL_callmeta`

[-0, +(0|1), *e*]

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

调用一个元方法。

如果在索引 `obj` 处的对象有元表， 且元表有域 `e` 。 这个函数会以该对象为参数调用这个域。 这种情况下，函数返回真并将调用返回值压栈。 如果那个位置没有元表，或没有对应的元方法， 此函数返回假（并不会将任何东西压栈）。

`luaL_checkany`

[-0, +0, *v*]

```
void luaL_checkany (lua_State *L, int arg);
```

检查函数在 `arg` 位置是否有任何类型（包括 **nil**）的参数。

`luaL_checkinteger`

[-0, +0, *v*]

```
lua_Integer luaL_checkinteger (lua_State *L, int arg);
```

检查函数的第 `arg` 个参数是否是一个 整数（或是可以被转换为一个整数） 并以 `lua_Integer` 类型返回这个整数值。

`luaL_checklstring`

[-0, +0, *v*]

```
const char *luaL_checklstring (lua_State *L, int arg, size_t *l);
```

检查函数的第 `arg` 个参数是否是一个 字符串，并返回该字符串； 如果 `l` 不为 `NULL` ， 将字符串的长度填入 `*l` 。

这个函数使用 `lua_tolstring` 来获取结果。 所以该函数有可能引发的转换都同样有效。

luaL_checknumber

[-0, +0, *v*]

```
lua_Number luaL_checknumber (lua_State *L, int arg);
```

检查函数的第 `arg` 个参数是否是一个 数字，并返回这个数字。

luaL_checkoption

[-0, +0, *v*]

```
int luaL_checkoption (lua_State *L,
                      int arg,
                      const char *def,
                      const char *const lst[]);
```

检查函数的第 `arg` 个参数是否是一个 字符串，并在数组 `lst` （比如是零结尾的字符串数组） 中查找这个字符串。 返回匹配到的字符串在数组中的索引号。 如果参数不是字符串，或是字符串在数组中匹配不到，都将抛出错误。

如果 `def` 不为 `NULL`， 函数就把 `def` 当作默认值。 默认值在参数 `arg` 不存在，或该参数是 **nil** 时生效。

这个函数通常用于将字符串映射为 C 枚举量。 （在 Lua 库中做这个转换可以让其使用字符串，而不是数字来做一些选项。）

luaL_checkstack

[-0, +0, *v*]

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

将栈空间扩展到 `top + sz` 个元素。 如果扩展不了，则抛出一个错误。 `msg` 是用于错误消息的额外文本 （ `NULL` 表示不需要额外文本）。

luaL_checkstring

[-0, +0, *v*]

```
const char *luaL_checkstring (lua_State *L, int arg);
```

检查函数的第 `arg` 个参数是否是一个 字符串并返回这个字符串。

这个函数使用 `lua_tolstring` 来获取结果。 所以该函数有可能引发的转换都同样有效。

`luaL_checktype`

[-0, +0, *v*]

```
void luaL_checktype (lua_State *L, int arg, int t);
```

检查函数的第 `arg` 个参数的类型是否是 `t` 。 参见 `lua_type` 查阅类型 `t` 的编码。

`luaL_checkudata`

[-0, +0, *v*]

```
void *luaL_checkudata (lua_State *L, int arg, const char *tname);
```

检查函数的第 `arg` 个参数是否是一个类型为 `tname` 的用户数据 （参见 `luaL_newmetatable`)。 它会返回该用户数据的地址 （参见 `lua_touserdata` ）。

`luaL_checkversion`

[-0, +0, –]

```
void luaL_checkversion (lua_State *L);
```

检查调用它的内核是否是创建这个 Lua 状态机的内核。 以及调用它的代码是否使用了相同的 Lua 版本。 同时也检查调用它的内核与创建该 Lua 状态机的内核 是否使用了同一片地址空间。

`luaL_dofile`

[-0, +?, *e*]

```
int luaL_dofile (lua_State *L, const char *filename);
```

加载并运行指定的文件。 它是用下列宏定义出来：

```
     (luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

如果没有错误，函数返回假； 有错则返回真。

`luaL_dostring`

[-0, +?, –]

```
int luaL_dostring (lua_State *L, const char *str);
```

加载并运行指定的字符串。 它是用下列宏定义出来：

```
     (luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

如果没有错误，函数返回假； 有错则返回真。

`luaL_error`

[-0, +0, *v*]

```
int luaL_error (lua_State *L, const char *fmt, ...);
```

抛出一个错误。 错误消息的格式由 `fmt` 给出。 后面需提供若干参数， 这些参数遵循 `lua_pushfstring` 中的规则。 如果能获得相关信息，它还会在消息前面加上错误发生时的文件名及行号。

这个函数永远不会返回。 但是在 C 函数中通常遵循惯用法： `return luaL_error(_args_)` 。

`luaL_execresult`

[-0, +3, *e*]

```
int luaL_execresult (lua_State *L, int stat);
```

这个函数用于生成标准库中和进程相关函数的返回值。 （指 `os.execute` 和 `io.close` ） 。

`luaL_fileresult`

[-0, +(1|3), *e*]

```
int luaL_fileresult (lua_State *L, int stat, const char *fname);
```

这个函数用于生成标准库中和文件相关的函数的返回值。 （指 (`io.open` ， `os.rename` ， `file:seek` ， 等。)。

luaL_getmetafield

[-0, +(0|1), *e*]

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

将索引 `obj` 处对象的元表中 `e` 域的值压栈。 如果该对象没有元表，或是该元表没有相关域， 此函数什么也不会压栈并返回 `LUA_TNIL` 。

luaL_getmetatable

[-0, +1, –]

```
int luaL_getmetatable (lua_State *L, const char *tname);
```

将注册表中 `tname` 对应的元表 （参见 `luaL_newmetatable`） 压栈。 如果没有 `tname` 对应的元表， 则将 **nil** 压栈并返回假。

luaL_getsubtable

[-0, +1, *e*]

```
int luaL_getsubtable (lua_State *L, int idx, const char *fname);
```

确保 `t[fname]` 是一张表，并将这张表压栈。 这里的 `t` 指索引 `idx` 处的值。 如果它原来就是一张表，返回真； 否则为它创建一张新表，返回假。

luaL_gsub

[-0, +1, *e*]

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```

将字符串 `s` 生成一个副本， 并将其中的所有字符串 `p` 都替换为字符串 `r` 。 将结果串压栈并返回它。

luaL_len

[-0, +0, *e*]

```
lua_Integer luaL_len (lua_State *L, int index);
```

以数字形式返回给定索引处值的“长度”； 它等价于在 Lua 中调用 '`#`' 的操作 （参见 §3.4.7）。 如果操作结果不是一个整数，则抛出一个错误。 （这种情况只发生在触发元方法时。）

`luaL_loadbuffer`

[-0, +1, –]

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```

等价于 `luaL_loadbufferx`， 其 `mode` 参数等于 `NULL`。

`luaL_loadbufferx`

[-0, +1, –]

```
int luaL_loadbufferx (lua_State *L,
                      const char *buff,
                      size_t sz,
                      const char *name,
                      const char *mode);
```

把一段缓存加载为一个 Lua 代码块。 这个函数使用 `lua_load` 来加载 `buff` 指向的长度为 `sz` 的内存区。

这个函数和 `lua_load` 返回值相同。 `name` 作为代码块的名字，用于调试信息和错误消息。 `mode` 字符串的作用同函数 `lua_load`。

`luaL_loadfile`

[-0, +1, *e*]

```
int luaL_loadfile (lua_State *L, const char *filename);
```

等价于 `luaL_loadfilex`， 其 `mode` 参数等于 `NULL`。

`luaL_loadfilex`

[-0, +1, *e*]

```
int luaL_loadfilex (lua_State *L, const char *filename,
                                            const char *mode);
```

把一个文件加载为 Lua 代码块。 这个函数使用 `lua_load` 加载文件中的数据。 代码块的名字被命名为 `filename` 。 如果 `filename` 为 `NULL` ， 它从标准输入加载。 如果文件的第一行以 `#` 打头，则忽略这一行。

`mode` 字符串的作用同函数 `lua_load` 。

此函数的返回值和 `lua_load` 相同， 不过它还可能产生一个叫做 `LUA_ERRFILE` 的出错码。这种错误发生于无法打开或读入文件时，或是文件的模式错误。

和 `lua_load` 一样，这个函数仅加载代码块不运行。

`luaL_loadstring`

[-0, +1, –]

```
int luaL_loadstring (lua_State *L, const char *s);
```

将一个字符串加载为 Lua 代码块。 这个函数使用 `lua_load` 加载一个零结尾的字符串 `s` 。

此函数的返回值和 `lua_load` 相同。

也和 `lua_load` 一样，这个函数仅加载代码块不运行。

`luaL_newlib`

[-0, +1, *e*]

```
void luaL_newlib (lua_State *L, const luaL_Reg l[]);
```

创建一张新的表，并把列表 `l` 中的函数注册进去。

它是用下列宏实现的：

```
     (luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

数组 `l` 必须是一个数组，而不能是一个指针。

`luaL_newlibtable`

[-0, +1, *e*]

```
void luaL_newlibtable (lua_State *L, const luaL_Reg l[]);
```

创建一张新的表，并预分配足够保存下数组 `l` 内容的空间（但不填充）。 这是给 `luaL_setfuncs` 一起用的 （参见 `luaL_newlib` ）。

它以宏形式实现， 数组 `l` 必须是一个数组，而不能是一个指针。

`luaL_newmetatable`

[-0, +1, *e*]

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

如果注册表中已存在键 `tname` ，返回 0 。 否则， 为用户数据的元表创建一张新表。 向这张表加入 `__name = tname` 键值对， 并将 `[tname] = new table` 添加到注册表中， 返回 1 。 （ `__name` 项可用于一些错误输出函数。）

这两种情况都会把最终的注册表中关联 `tname` 的值压栈。

`luaL_newstate`

[-0, +0, –]

```
lua_State *luaL_newstate (void);
```

创建一个新的 Lua 状态机。 它以一个基于标准 C 的 `realloc` 函数实现的内存分配器 调用 `lua_newstate` 。 并把可打印一些出错信息到标准错误输出的 panic 函数（参见 §4.6） 设置好，用于处理致命错误。

返回新的状态机。 如果内存分配失败，则返回 `NULL` 。

`luaL_openlibs`

[-0, +0, *e*]

```
void luaL_openlibs (lua_State *L);
```

打开指定状态机中的所有 Lua 标准库。

`luaL_optinteger`

[-0, +0, *v*]

```
lua_Integer luaL_optinteger (lua_State *L,
                             int arg,
                             lua_Integer d);
```

如果函数的第 `arg` 个参数是一个 整数（或可以转换为一个整数）， 返回该整数。 若该参数不存在或是 **nil**， 返回 `d` 。 除此之外的情况，抛出错误。

`luaL_optlstring`

[-0, +0, *v*]

```
const char *luaL_optlstring (lua_State *L,
                             int arg,
                             const char *d,
                             size_t *l);
```

如果函数的第 `arg` 个参数是一个 字符串，返回该字符串。 若该参数不存在或是 **nil**， 返回 `d` 。 除此之外的情况，抛出错误。

若 `l` 不为 `NULL` ， 将结果的长度填入 `*l` 。

`luaL_optnumber`

[-0, +0, *v*]

```
lua_Number luaL_optnumber (lua_State *L, int arg, lua_Number d);
```

如果函数的第 `arg` 个参数是一个 数字，返回该数字。 若该参数不存在或是 **nil**， 返回 `d` 。 除此之外的情况，抛出错误。

`luaL_optstring`

[-0, +0, *v*]

```
const char *luaL_optstring (lua_State *L,
                            int arg,
                            const char *d);
```

如果函数的第 `arg` 个参数是一个 字符串，返回该字符串。 若该参数不存在或是 **nil**， 返回 `d` 。 除此之外的情况，抛出错误。

`luaL_prepbuffer`

[-?, +?, *e*]

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

等价于 `luaL_prepbuffsize` ， 其预定义大小为 `LUAL_BUFFERSIZE` 。

`luaL_prepbuffsize`

[-?, +?, *e*]

```
char *luaL_prepbuffsize (luaL_Buffer *B, size_t sz);
```

返回一段大小为 `sz` 的空间地址。 你可以将字符串复制其中以加到缓存 `B` 内 （参见 `luaL_Buffer` ）。 将字符串复制其中后，你必须调用 `luaL_addsize` 传入字符串的大小，才会真正把它加入缓存。

`luaL_pushresult`

[-?, +1, *e*]

```
void luaL_pushresult (luaL_Buffer *B);
```

结束对缓存 `B` 的使用，将最终的字符串留在栈顶。

`luaL_pushresultsize`

[-?, +1, *e*]

```
void luaL_pushresultsize (luaL_Buffer *B, size_t sz);
```

等价于 `luaL_addsize` ， `luaL_pushresult` 。

`luaL_ref`

[-1, +0, *e*]

```
int luaL_ref (lua_State *L, int t);
```

针对栈顶的对象，创建并返回一个在索引 `t` 指向的表中的 引用 （最后会弹出栈顶对象）。

此引用是一个唯一的整数键。 只要你不向表 `t` 手工添加整数键， `luaL_ref` 可以保证它返回的键的唯一性。 你可以通过调用 `lua_rawgeti(L, t, r)` 来找回由 `r` 引用的对象。 函数 `luaL_unref` 用来释放一个引用关联的对象

如果栈顶的对象是 **nil**， `luaL_ref` 将返回常量 `LUA_REFNIL` 。 常量 `LUA_NOREF` 可以保证和 `luaL_ref` 能返回的其它引用值不同。

`luaL_Reg`

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

用于 `luaL_setfuncs` 注册函数的数组类型。 `name` 指函数名， `func` 是函数指针。 任何 `luaL_Reg` 数组必须以一对 `name` 与 `func` 皆为 `NULL` 结束。

`luaL_requiref`

[-0, +1, *e*]

```
void luaL_requiref (lua_State *L, const char *modname,
                    lua_CFunction openf, int glb);
```

如果 `modname` 不在 `package.loaded` 中， 则调用函数 `openf` ，并传入字符串 `modname` 。 将其返回值置入 `package.loaded[modname]` 。 这个行为好似该函数通过 `require` 调用过一样。

如果 `glb` 为真， 同时也讲模块设到全局变量 `modname` 里。

在栈上留下该模块的副本。

`luaL_setfuncs`

[-nup, +0, *e*]

```
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);
```

把数组 `l` 中的所有函数 （参见 `luaL_Reg` ） 注册到栈顶的表中（该表在可选的上值之下， 见下面的解说）。

若 `nup` 不为零， 所有的函数都共享 `nup` 个上值。 这些值必须在调用之前， 压在表之上。 这些值在注册完毕后都会从栈弹出。

`luaL_setmetatable`

[-0, +0, –]

```
void luaL_setmetatable (lua_State *L, const char *tname);
```

将注册表中 `tname` 关联元表 （参见 `luaL_newmetatable` ） 设为栈顶对象的元表。

`luaL_Stream`

```
typedef struct luaL_Stream {
  FILE *f;
  lua_CFunction closef;
} luaL_Stream;
```

标准输入输出库中用到的标准文件句柄结构。

文件句柄实现为一个完全用户数据， 其元表被称为 `LUA_FILEHANDLE` （ `LUA_FILEHANDLE` 是一个代表真正元表的名字的宏） 。 这张元表由标准输入输出库（参见 `luaL_newmetatable` ） 创建。

用户数据必须以结构 `luaL_Stream` 开头； 此结构其后可以包含任何其它数据。 `f` 域指向一个 C 数据流 （如果它为 `NULL` 表示一个没有创建好的句柄） 。 `closef` 域指向一个在关闭或回收该流时需要调用的 Lua 函数。 该函数将收到一个参数， 即文件句柄。 它需要返回 **true**（操作成功）或 **nil** 加错误消息（出错的时候） 。 一旦 Lua 调用过这个域， 该域的值就会修改为 `NULL` 以提示这个句柄已经被关闭了。

`luaL_testudata`

[-0, +0, *e*]

```
void *luaL_testudata (lua_State *L, int arg, const char *tname);
```

此函数和 `luaL_checkudata` 类似。 但它在测试失败时会返回 `NULL` 而不是抛出错误。

`luaL_tolstring`

[-0, +1, *e*]

```
const char *luaL_tolstring (lua_State *L, int idx, size_t *len);
```

将给定索引处的 Lua 值转换为一个相应格式的 C 字符串。 结果串不仅会压栈， 还会由函数返回。 如果 `len` 不为 `NULL` ， 它还把字符串长度设到 `*len` 中。

如果该值有一个带 `"__tostring"` 域的元表， `luaL_tolstring` 会以该值为参数去调用对应的元方法， 并将其返回值作为结果。

`luaL_traceback`

[-0, +1, *e*]

```
void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,
                     int level);
```

将栈 `L1` 的栈回溯信息压栈。 如果 `msg` 不为 `NULL` ，它会附加到栈回溯信息之前。`level` 参数指明从第几层开始做栈回溯。

luaL_typename

[-0, +0, –]

```
const char *luaL_typename (lua_State *L, int index);
```

返回给定索引处值的类型名。

luaL_unref

[-0, +0, –]

```
void luaL_unref (lua_State *L, int t, int ref);
```

释放索引 `t` 处表的 `ref` 引用对象 （参见 `luaL_ref` ）。 此条目会从表中移除以让其引用的对象可被垃圾收集。 而引用 `ref` 也被回收再次使用。

如果 `ref` 为 `LUA_NOREF` 或 `LUA_REFNIL` ， `luaL_unref` 什么也不做。

luaL_where

[-0, +1, *e*]

```
void luaL_where (lua_State *L, int lvl);
```

将一个用于表示 `lvl` 层栈的控制点位置的字符串压栈。 这个字符串遵循下面的格式：

```
     _chunkname_:_currentline_:
```

0 层指当前正在运行的函数， 1 层指调用正在运行函数的函数， 依次类推。

这个函数用于构建错误消息的前缀。

6 – 标准库

标准库提供了一些有用的函数, 它们都是直接用 C API 实现的。 其中一些函数提供了原本语言就有的服务 （例如， `type` 与 `getmetatable` ）； 另一些提供和“外部”打交道的服务（例如 I/O ）； 还有些本可以用 Lua 本身来实现，但在 C 中实现可以满足关键点上的性能需求 （例如 `table.sort` ）。

所有的库都是直接用 C API 实现的，并以分离的 C 模块形式提供。 目前，Lua 有下列标准库：

- 基础库 (§6.1);
- 协程库 (§6.2);
- 包管理库 (§6.3);
- 字符串控制 (§6.4);
- 基础 UTF-8 支持 (§6.5);
- 表控制 (§6.6);
- 数学函数 (§6.7) (sin ， log 等);
- 输入输出 (§6.8);
- 操作系统库 (§6.9);
- 调试库 (§6.10).

除了基础库和包管理库， 其它库都把自己的函数放在一张全局表的域中， 或是以对象方法的形式提供。

要使用这些库， C 的宿主程序需要先调用一下 `luaL_openlibs` 这个函数， 这样就能打开所有的标准库。 或者宿主程序也可以用 `luaL_requiref` 分别打开这些库： `luaopen_base` （基础库）， `luaopen_package` （包管理库）， `luaopen_coroutine` （协程库）， `luaopen_string` （字符串库）， `luaopen_utf8` （UTF8 库）， `luaopen_table` （表处理库）， `luaopen_math` （数学库）， `luaopen_io` （I/O 库）， `luaopen_os` （操作系统库）， `luaopen_debug` （调试库）。 这些函数都定义在 `lualib.h` 中。

6.1 – 基础函数

基础库提供了 Lua 核心函数。 如果你不将这个库包含在你的程序中， 你就需要小心检查程序是否需要自己提供其中一些特性的实现。

`assert (v [, message])`

如果其参数 `v` 的值为假（**nil** 或 **false**）， 它就调用 `error` ； 否则，返回所有的参数。 在错误情况时， `message` 指那个错误对象； 如果不提供这个参数，参数默认为 "`assertion failed!`" 。

`collectgarbage ([opt [, arg]])`

这个函数是垃圾收集器的通用接口。 通过参数 `opt` 它提供了一组不同的功能：

- "**`collect`**": 做一次完整的垃圾收集循环。 这是默认选项。
- "**`stop`**": 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。
- "**`restart`**": 重启垃圾收集器的自动运行。
- "**`count`**": 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。
- "**`step`**": 单步运行垃圾收集器。 步长“大小”由 `arg` 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 **true** 。
- "**`setpause`**": 将 `arg` 设为收集器的 间歇率 （参见 §2.5）。 返回 间歇率 的前一个值。
- "**`setstepmul`**": 将 `arg` 设为收集器的 步进倍率 （参见 §2.5）。 返回 步进倍率 的前一个值。
- "**`isrunning`**": 返回表示收集器是否在工作的布尔值 （即未被停止）。

`dofile ([filename])`

打开该名字的文件，并执行文件中的 Lua 代码块。 不带参数调用时， `dofile` 执行标准输入的内容（ `stdin` ）。 返回该代码块的所有返回值。 对于有错误的情况， `dofile` 将错误反馈给调用者 （即， `dofile` 没有运行在保护模式下）。

`error (message [, level])`

中止上一次保护函数调用， 将错误对象 `message` 返回。 函数 `error` 永远不会返回。

当 message 是一个字符串时，通常 `error` 会把一些有关出错位置的信息附加在消息的前头。 `level` 参数指明了怎样获得出错位置。 对于 level 1 （默认值）， 出错位置指 `error` 函数调用的位置。 Level 2 将出错位置指向调用 `error` 的函数的函数；以此类推。 传入 level 0 可以避免在消息前添加出错位置信息。

`_G`

一个全局变量（非函数）， 内部储存有全局环境（参见 §2.2）。 Lua 自己不使用这个变量； 改变这个变量的值不会对任何环境造成影响，反之亦然。

`getmetatable (object)`

如果 `object` 不包含元表，返回 **nil** 。 否则，如果在该对象的元表中有 `"__metatable"` 域时返回其关联值， 没有时返回该对象的元表。

`ipairs (t)`

返回三个值（迭代函数、表 `t` 以及 0 ）， 如此，以下代码

```
for i,v in ipairs(t) do _body_ end
```

将迭代键值对（ `1,t[1]` ），(`2,t[2]`)， ... ， 直到第一个空值。

`load (chunk [, chunkname [, mode [, env]]])`

加载一个代码块。

如果 `chunk` 是一个字符串，代码块指这个字符串。 如果 `chunk` 是一个函数， `load` 不断地调用它获取代码块的片断。 每次对 `chunk` 的调用都必须返回一个字符串紧紧连接在上次调用的返回串之后。 当返回空串、**nil**、或是不返回值时，都表示代码块结束。

如果没有语法错误， 则以函数形式返回编译好的代码块； 否则，返回 **nil** 加上错误消息。

如果结果函数有上值， `env` 被设为第一个上值。 若不提供此参数，将全局环境替代它。 所有其它上值初始化为 **nil**。 （当你加载主代码块时候，结果函数一定有且仅有一个上值 `_ENV` （参见 §2.2））。 然而，如果你加载一个用函数（参见 `string.dump`， 结果函数可以有任意数量的上值） 创建出来的二进制代码块时，所有的上值都是新创建出来的。 也就是说它们不会和别的任何函数共享。

`chunkname` 在错误消息和调试消息中（参见 §4.9），用于代码块的名字。 如果不提供此参数，它默认为字符串`chunk` 。 `chunk` 不是字符串时，则为 " `=(load)` " 。

字符串 `mode` 用于控制代码块是文本还是二进制（即预编译代码块）。 它可以是字符串 " `b` " （只能是二进制代码块）， " `t` " （只能是文本代码块）， 或 " `bt` " （可以是二进制也可以是文本）。 默认值为 " `bt` "。

Lua 不会对二进制代码块做健壮性检查。 恶意构造一个二进制块有可能把解释器弄崩溃。

`loadfile ([filename [, mode [, env]]])`

和 `load` 类似， 不过是从文件 `filename` 或标准输入（如果文件名未提供）中获取代码块。

`next (table [, index])`

运行程序来遍历表中的所有域。 第一个参数是要遍历的表，第二个参数是表中的某个键。 `next` 返回该键的下一个键及其关联的值。 如果用 **nil** 作为第二个参数调用 `next` 将返回初始键及其关联值。 当以最后一个键去调用，或是以 **nil** 调用一张空表时， `next` 返回 **nil**。 如果不提供第二个参数，将认为它就是 **nil**。 特别指出，你可以用 `next(t)` 来判断一张表是否是空的。

索引在遍历过程中的次序无定义， 即使是数字索引也是这样。 （如果想按数字次序遍历表，可以使用数字形式的 **for** 。）

当在遍历过程中你给表中并不存在的域赋值， `next` 的行为是未定义的。 然而你可以去修改那些已存在的域。 特别指出，你可以清除一些已存在的域。

`pairs (t)`

如果 `t` 有元方法 `__pairs`， 以 `t` 为参数调用它，并返回其返回的前三个值。

否则，返回三个值： `next` 函数， 表 `t`，以及 **nil**。 因此以下代码

```
for k,v in pairs(t) do _body_ end
```

能迭代表 `t` 中的所有键值对。

参见函数 `next` 中关于迭代过程中修改表的风险。

`pcall (f [, arg1, ···])`

传入参数，以 保护模式 调用函数 `f` 。 这意味着 `f` 中的任何错误不会抛出； 取而代之的是， `pcall` 会将错误捕获到，并返回一个状态码。 第一个返回值是状态码（一个布尔量）， 当没有错误时，其为真。 此时， `pcall` 同样会在状态码后返回所有调用的结果。 在有错误时， `pcall` 返回 **false** 加错误消息。

`print (···)`

接收任意数量的参数，并将它们的值打印到 `stdout` 。 它用 `tostring` 函数将每个参数都转换为字符串。 `print` 不用于做格式化输出。仅作为看一下某个值的快捷方式。 多用于调试。 完整的对输出的控制，请使用 `string.format` 以及 `io.write` 。

`rawequal (v1, v2)`

在不触发任何元方法的情况下 检查 `v1` 是否和 `v2` 相等。 返回一个布尔量。

`rawget (table, index)`

在不触发任何元方法的情况下 获取 `table[index]` 的值。 `table` 必须是一张表； `index` 可以是任何值。

`rawlen (v)`

在不触发任何元方法的情况下 返回对象 `v` 的长度。 `v` 可以是表或字符串。 它返回一个整数。

`rawset (table, index, value)`

在不触发任何元方法的情况下 将 `table[index]` 设为 `value` 。 `table` 必须是一张表， `index` 可以是 **nil** 与 NaN 之外的任何值。 `value` 可以是任何 Lua 值。

这个函数返回 `table` 。

`select (index, ···)`

如果 `index` 是个数字， 那么返回参数中第 `index` 个之后的部分； 负的数字会从后向前索引（-1 指最后一个参数）。 否则， `index` 必须是字符串 `"#"` ， 此时 `select` 返回参数的个数。

`setmetatable (table, metatable)`

给指定表设置元表。 （你不能在 Lua 中改变其它类型值的元表，那些只能在 C 里做。） 如果 `metatable` 是 **nil**， 将指定表的元表移除。 如果原来那张元表有 `"__metatable"` 域，抛出一个错误。

这个函数返回 `table` 。

`tonumber (e [, base])`

如果调用的时候没有 `base` ， `tonumber` 尝试把参数转换为一个数字。 如果参数已经是一个数字，或是一个可以转换为数字的字符串， `tonumber` 就返回这个数字； 否则返回 **nil**。

字符串的转换结果可能是整数也可能是浮点数， 这取决于 Lua 的转换文法（参见 §3.1）。 （字符串可以有前置和后置的空格，可以带符号。）

当传入 `base` 调用它时， `e` 必须是一个以该进制表示的整数字符串。 进制可以是 2 到 36 （包含 2 和 36）之间的任何整数。 大于 10 进制时，字母 '`A`' （大小写均可）表示 10 ， '`B`' 表示 11，依次到 '`Z`' 表示 35 。 如果字符串 `e` 不是该进制下的合法数字， 函数返回 **nil**。

`tostring (v)`

可以接收任何类型，它将其转换为人可阅读的字符串形式。 浮点数总被转换为浮点数的表现形式（小数点形式或是指数形式）。 （如果想完全控制数字如何被转换，可以使用 `string.format` 。）

如果 `v` 有 `"__tostring"` 域的元表， `tostring` 会以 `v` 为参数调用它。 并用它的结果作为返回值。

`type (v)`

将参数的类型编码为一个字符串返回。 函数可能的返回值有 "`nil`" （一个字符串，而不是 **nil** 值）， "`number`"， "`string`"， "`boolean`"， "`table`"， "`function`"， "`thread`"， "`userdata`"。

`_VERSION`

一个包含有当前解释器版本号的全局变量（并非函数）。 当前这个变量的值为 "`Lua 5.3`"。

`xpcall (f, msgh [, arg1, ···])`

这个函数和 `pcall` 类似。 不过它可以额外设置一个消息处理器 `msgh` 。

6.2 – 协程管理

关于协程的操作作为基础库的一个子库， 被放在一个独立表 `coroutine` 中。 协程的介绍参见 §2.6 。

`coroutine.create (f)`

创建一个主体函数为 `f` 的新协程。 `f` 必须是一个 Lua 的函数。 返回这个新协程，它是一个类型为 `"thread"` 的对象。

`coroutine.isyieldable ()`

如果正在运行的协程可以让出，则返回真。

不在主线程中或不在一个无法让出的 C 函数中时，当前协程是可让出的。

`coroutine.resume (co [, val1, ···])`

开始或继续协程 `co` 的运行。 当你第一次延续一个协程，它会从主体函数处开始运行。 `val1`, ... 这些值会以参数形式传入主体函数。 如果该协程被让出， `resume` 会重新启动它； `val1`, ... 这些参数会作为让出点的返回值。

如果协程运行起来没有错误， `resume` 返回 **true** 加上传给 `yield` 的所有值 （当协程让出）， 或是主体函数的所有返回值（当协程中止）。 如果有任何错误发生， `resume` 返回 **false** 加错误消息。

`coroutine.running ()`

返回当前正在运行的协程加一个布尔量。 如果当前运行的协程是主线程，其为真。

`coroutine.status (co)`

以字符串形式返回协程 `co` 的状态： 当协程正在运行（它就是调用 `status` 的那个） ，返回 `"running"` ； 如果协程调用 `yield` 挂起或是还没有开始运行，返回 `"suspended"` ； 如果协程是活动的，都并不在运行（即它正在延续其它协程） ，返回 `"normal"` ； 如果协程运行完主体函数或因错误停止，返回 `"dead"` 。

`coroutine.wrap (f)`

创建一个主体函数为 `f` 的新协程。 `f` 必须是一个 Lua 的函数。 返回一个函数， 每次调用该函数都会延续该协程。 传给这个函数的参数都会作为 `resume` 的额外参数。 和 `resume` 返回相同的值， 只是没有第一个布尔量。 如果发生任何错误，抛出这个错误。

`coroutine.yield (···)`

挂起正在调用的协程的执行。 传递给 `yield` 的参数都会转为 `resume` 的额外返回值。

6.3 – 模块

包管理库提供了从 Lua 中加载模块的基础库。 只有一个导出函数直接放在全局环境中： `require` 。 所有其它的部分都导出在表 `package` 中。

`require (modname)`

加载一个模块。 这个函数首先查找 `package.loaded` 表， 检测 `modname` 是否被加载过。 如果被加载过， `require` 返回 `package.loaded[modname]` 中保存的值。 否则，它试着为模块寻找 加载器 。

`require` 遵循 `package.searchers` 序列的指引来查找加载器。 如果改变这个序列，我们可以改变 `require` 如何查找一个模块。 下列说明基于 `package.searchers` 的默认配置。

首先 `require` 查找 `package.preload[modname]` 。 如果这里有一个值，这个值（必须是一个函数）就是那个加载器。 否则 `require` 使用 Lua 加载器去查找 `package.path` 的路径。 如果查找失败，接着使用 C 加载器去查找 `package.cpath` 的路径。 如果都失败了，再尝试 一

体化 加载器 （参见 `package.searchers` ）。

每次找到一个加载器， `require` 都用两个参数调用加载器： `modname` 和一个在获取加载器过程中得到的参数。 （如果通过查找文件得到的加载器，这个额外参数是文件名。） 如果加载器返回非空值， `require` 将这个值赋给 `package.loaded[modname]` 。 如果加载器没能返回一个非空值用于赋给 `package.loaded[modname]` ， `require` 会在那里设入 **true** 。 无论是什么情况， `require` 都会返回 `package.loaded[modname]` 的最终值。

如果在加载或运行模块时有错误， 或是无法为模块找到加载器， `require` 都会抛出错误。

`package.config`

一个描述有一些为包管理准备的编译期配置信息的串。 这个字符串由一系列行构成：

- 第一行是目录分割串。 对于 Windows 默认是 '`\`'， 对于其它系统是 '`/`' 。
- 第二行是用于路径中的分割符。默认值是 '`;`' 。
- 第三行是用于标记模板替换点的字符串。 默认是 '`?`' 。
- 第四行是在 Windows 中将被替换成执行程序所在目录的路径的字符串。 默认是 '`!`' 。
- 第五行是一个记号， 该记号之后的所有文本将在构建 `luaopen_` 函数名时被忽略掉。 默认是 '`-`'。

`package.cpath`

这个路径被 `require` 在 C 加载器中做搜索时用到。

Lua 用和初始化 Lua 路径 `package.path` 相同的方式初始化 C 路径 `package.cpath` 。 它会使用环境变量 `LUA_CPATH_5_3` 或 环境变量 `LUA_CPATH` 初始化。 要么就采用 `luaconf.h` 中定义的默认路径。

`package.loaded`

用于 `require` 控制哪些模块已经被加载的表。 当你请求一个 `modname` 模块， 且 `package.loaded[modname]` 不为假时， `require` 简单返回储存在内的值。

这个变量仅仅是对真正那张表的引用； 改变这个值并不会改变 `require` 使用的表。

`package.loadlib (libname, funcname)`

让宿主程序动态链接 C 库 `libname` 。

当 `funcname` 为 "`*`"， 它仅仅连接该库，让库中的符号都导出给其它动态链接库使用。 否则， 它查找库中的函数 `funcname` ， 以 C 函数的形式返回这个函数。 因此， `funcname` 必须遵循原型 `lua_CFunction` （参见 `lua_CFunction` ） 。

这是一个低阶函数。 它完全绕过了包模块系统。 和 `require` 不同， 它不会做任何路径查询，也不会自动加扩展名。 `libname` 必须是一个 C 库需要的完整的文件名，如果有必要，需要提供路径和扩展名。 `funcname` 必须是 C 库需要的准确名字 （这取决于使用的 C 编译器和链接器）。

这个函数在标准 C 中不支持。 因此，它只在部分平台有效 （ Windows ， Linux ， Mac OS X, Solaris, BSD, 加上支持 `dlfcn` 标准的 Unix 系统）。

`package.path`

这个路径被 `require` 在 Lua 加载器中做搜索时用到。

在启动时，Lua 用环境变量 `LUA_PATH_5_3` 或环境变量 `LUA_PATH` 来初始化这个变量。 或采用 `luaconf.h` 中的默认路径。 环境变量中出现的所有 " `;;` " 都会被替换成默认路径。

`package.preload`

保存有一些特殊模块的加载器 （参见 `require` ）。

这个变量仅仅是对真正那张表的引用； 改变这个值并不会改变 `require` 使用的表。

`package.searchers`

用于 `require` 控制如何加载模块的表。

这张表内的每一项都是一个 查找器函数。 当查找一个模块时， `require` 按次序调用这些查找器， 并传入模块名（ `require` 的参数）作为唯一的一个参数。 此函数可以返回另一个函数（模块的 加载器）加上另一个将传递给这个加载器的参数。 或是返回一个描述为何没有找到这个模块的字符串 （或是返回 **nil** 什么也不想说）。

Lua 用四个查找器函数初始化这张表。

第一个查找器就是简单的在 `package.preload` 表中查找加载器。

第二个查找器用于查找 Lua 库的加载库。 它使用储存在 `package.path` 中的路径来做查找工作。 查找过程和函数 `package.searchpath` 描述的一致。

第三个查找器用于查找 C 库的加载库。 它使用储存在 `package.cpath` 中的路径来做查找工作。 同样， 查找过程和函数 `package.searchpath` 描述的一致。 例如，如果 C 路径是这样一个字符串

```
     "./?.so;./?.dll;/usr/local/?/init.so"
```

查找器查找模块 `foo` 会依次尝试打开文件 `./foo.so` ， `./foo.dll` ， 以及 `/usr/local/foo/init.so` 。 一旦它找到一个 C 库， 查找器首先使用动态链接机制连接该库。 然后尝试在该库中找到可以用作加载器的 C 函数。 这个 C 函数的名字是 " `luaopen_` " 紧接模块名的字符串， 其中字符串中所有的下划线都会被替换成点。 此外，如果模块名中有横线， 横线后面的部分（包括横线）都被去掉。 例如，如果模块名为 `a.b.c-v2.1` ， 函数名就是 `luaopen_a_b_c` 。

第四个搜索器是 一体化加载器。 它从 C 路径中查找指定模块的根名字。 例如，当请求 `a.b.c` 时， 它将查找 `a` 这个 C 库。 如果找得到，它会在里面找子模块的加载函数。 在我们的例子中，就是找 `luaopen_a_b_c` 。 利用这个机制，可以把若干 C 子模块打包进单个库。 每个子模块都可以有原本的加载函数名。

除了第一个（预加载）搜索器外，每个搜索器都会返回 它找到的模块的文件名。 这和 `package.searchpath` 的返回值一样。 第一个搜索器没有返回值。

`package.searchpath (name, path [, sep [, rep]])`

在指定 `path` 中搜索指定的 `name` 。

路径是一个包含有一系列以分号分割的 模板 构成的字符串。 对于每个模板，都会用 `name` 替换其中的每个问号（如果有的话）。 且将其中的 `sep` （默认是点）替换为 `rep` （默认是系统的目录分割符）。 然后尝试打开这个文件名。

例如，如果路径是字符串

```
     "./?.lua;./?.lc;/usr/local/?/init.lua"
```

搜索 `foo.a` 这个名字将 依次尝试打开文件 `./foo/a.lua` ， `./foo/a.lc` ，以及 `/usr/local/foo/a/init.lua` 。

返回第一个可以用读模式打开（并马上关闭该文件）的文件的名字。 如果不存在这样的文件，返回 **nil** 加上错误消息。 （这条错误消息列出了所有尝试打开的文件名。）

6.4 – 字符串处理

这个库提供了字符串处理的通用函数。 例如字符串查找、子串、模式匹配等。 当在 Lua 中对字符串做索引时，第一个字符从 1 开始计算（而不是 C 里的 0 ）。 索引可以是负数，它指从字符串末尾反向解析。 即，最后一个字符在 -1 位置处，等等。

字符串库中的所有函数都在表 `string` 中。 它还将其设置为字符串元表的 `__index` 域。 因此，你可以以面向对象的形式使用字符串函数。 例如， `string.byte(s,i)` 可以写成 `s:byte(i)` 。

字符串库假定采用单字节字符编码。

`string.byte (s [, i [, j]])`

返回字符 `s[i]` ， `s[i+1]` ， ... ， `s[j]` 的内部数字编码。 `i` 的默认值是 1 ； `j` 的默认值是 `i` 。 这些索引以函数 `string.sub` 的规则修正。

数字编码没有必要跨平台。

`string.char (···)`

接收零或更多的整数。 返回和参数数量相同长度的字符串。 其中每个字符的内部编码值等于对应的参数值。

数字编码没有必要跨平台。

`string.dump (function [, strip])`

返回包含有以二进制方式表示的（一个 二进制代码块 ）指定函数的字符串。 之后可以用 `load` 调用这个字符串获得 该函数的副本（但是绑定新的上值）。 如果 `strip` 为真值， 二进制代码块不携带该函数的调试信息 （局部变量名，行号，等等。）。

带上值的函数只保存上值的数目。 当（再次）加载时，这些上值被更新为 **nil** 的实例。 （你可以使用调试库按你需要的方式来序列化上值，并重载到函数中）

`string.find (s, pattern [, init [, plain]])`

查找第一个字符串 `s` 中匹配到的 `pattern` （参见 §6.4.1）。 如果找到一个匹配， `find` 会返回 `s` 中关于它起始及终点位置的索引； 否则，返回 **nil**。 第三个可选数字参数 `init` 指明从哪里开始搜索； 默认值为 1 ，同时可以是负值。 第四个可选参数 `plain` 为 **true** 时， 关闭模式匹配机制。 此时函数仅做直接的 “查找子串”的操作， 而 `pattern` 中没有字符被看作魔法字符。 注意，如果给定了 `plain` ，就必须写上 `init` 。

如果在模式中定义了捕获，捕获到的若干值也会在两个索引之后返回。

`string.format (formatstring, ···)`

返回不定数量参数的格式化版本， 格式化串为第一个参数（必须是一个字符串）。 格式化字符串遵循 ISO C 函数 `sprintf` 的规则。 不同点在于选项 `*` ， `h` ， `L` ， `l` ， `n` ， `p` 不支持， 另外还增加了一个选项 `q` 。 `q` 选项将一个字符串格式化为两个双引号括起，对内部字符做恰当的转义处理的字符串。 该字符串可以安全的被 Lua 解释器读回来。 例如， 调用

```
string.format('%q', 'a string with "quotes" and \n new line')
```

会产生字符串：

```
"a string with \"quotes\" and \
 new line"
```

选项 `A` and `a` （如果有的话）， `E` , `e` , `f` , `G` , and `g` 都期待一个对应的数字参数。选项 `c` , `d` , `i` , `o` , `u` , `X` , and `x` 则期待一个整数。 选项 `q` 期待一个字符串； 选项 `s` 期待一个没有内嵌零的字符串。 如果选项 `s` 对应的参数不是字符串，它会用和 `tostring` 一致的规则转换成字符串。

`string.gmatch (s, pattern)`

返回一个迭代器函数。 每次调用这个函数都会继续以 `pattern` （参见 §6.4.1） 对 `s` 做匹配，并返回所有捕获到的值。 如果 `pattern` 中没有指定捕获，则每次捕获整个 `pattern` 。

下面这个例子会循环迭代字符串 `s` 中所有的单词， 并逐行打印：

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

下一个例子从指定的字符串中收集所有的键值对 `key=value` 置入一张表：

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

对这个函数来说，模板前开始的 ' `^` ' 不会当成锚点。因为这样会阻止迭代。

`string.gsub (s, pattern, repl [, n])`

将字符串 `s` 中，所有的（或是在 `n` 给出时的前 `n` 个） `pattern` （参见 §6.4.1）都替换成 `repl` ，并返回其副本。 `repl` 可以是字符串、表、或函数。 `gsub` 还会在第二个返回值返回一共发生了多少次匹配。 `gsub` 这个名字来源于 *Global SUBstitution* 。

如果 `repl` 是一个字符串，那么把这个字符串作为替换品。 字符 `%` 是一个转义符： `repl` 中的所有形式为 `%_d_` 的串表示 第 *d* 个捕获到的子串，*d* 可以是 1 到 9 。 串 `%0` 表示整个匹配。 串 `%%` 表示单个 `%` 。

如果 `repl` 是张表，每次匹配时都会用第一个捕获物作为键去查这张表。

如果 `repl` 是个函数，则在每次匹配发生时都会调用这个函数。 所有捕获到的子串依次作为参数传入。

任何情况下，模板中没有设定捕获都看成是捕获整个模板。

如果表的查询结果或函数的返回结果是一个字符串或是个数字，都将其作为替换用串；而在返回 **false** 或 **nil** 时不作替换（即保留匹配前的原始串）。

这里有一些用例：

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.3"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.3.tar.gz"
```

`string.len (s)`

接收一个字符串，返回其长度。空串 `""` 的长度为 0 。内嵌零也统计在内，因此 `"a\000bc\000"` 的长度为 5 。

`string.lower (s)`

接收一个字符串，将其中的大写字符都转为小写后返回其副本。其它的字符串不会更改。对大写字符的定义取决于当前的区域设置。

`string.match (s, pattern [, init])`

在字符串 `s` 中找到第一个能用 `pattern` （参见 §6.4.1）匹配到的部分。如果能找到，`match` 返回其中的捕获物；否则返回 **nil** 。如果 `pattern` 中未指定捕获，返回整个 `pattern` 捕获到的串。第三个可选数字参数 `init` 指明从哪里开始搜索；它默认为 1 且可以是负数。

`string.pack (fmt, v1, v2, ···)`

返回一个打包了（即以二进制形式序列化） `v1`，`v2` 等值的二进制字符串。字符串 `fmt` 为打包格式（参见 §6.4.2）。

`string.packsize (fmt)`

返回以指定格式用 `string.pack` 打包的字符串的长度。 格式化字符串中不可以有变长选项 '`s`' 或 '`z`' （参见 §6.4.2）。

`string.rep (s, n [, sep])`

返回 `n` 个字符串 `s` 以字符串 `sep` 为分割符连在一起的字符串。 默认的 `sep` 值为空字符串（即没有分割符）。 如果 `n` 不是正数则返回空串。

`string.reverse (s)`

返回字符串 `s` 的翻转串。

`string.sub (s, i [, j])`

返回 `s` 的子串， 该子串从 `i` 开始到 `j` 为止； `i` 和 `j` 都可以为负数。 如果不给出 `j` ，就当它是 -1 （和字符串长度相同）。 特别是， 调用 `string.sub(s,1,j)` 可以返回 `s` 的长度为 `j` 的前缀串， 而 `string.sub(s, -i)` 返回长度为 `i` 的后缀串。

如果在对负数索引转义后 `i` 小于 1 的话，就修正回 1 。 如果 `j` 比字符串的长度还大，就修正为字符串长度。 如果在修正之后， `i` 大于 `j`， 函数返回空串。

`string.unpack (fmt, s [, pos])`

返回以格式 `fmt` （参见 §6.4.2） 打包在字符串 `s` （参见 `string.pack`） 中的值。 选项 `pos` （默认为 1 ）标记了从 `s` 中哪里开始读起。 读完所有的值后，函数返回 `s` 中第一个未读字节的位置。

`string.upper (s)`

接收一个字符串，将其中的小写字符都转为大写后返回其副本。 其它的字符串不会更改。 对小写字符的定义取决于当前的区域设置。

6.4.1 – 匹配模式

Lua 中的匹配模式直接用常规的字符串来描述。 它用于模式匹配函数 `string.find`, `string.gmatch`, `string.gsub`, `string.match`。 这一节表述了这些字符串的语法及含义（即它能匹配到什么）。

字符类：

字符类 用于表示一个字符集合。 下列组合可用于字符类：

- ***x*:** （这里 *x* 不能是 魔法字符 `^$()%.[]*+-?` 中的一员） 表示字符 *x* 自身。
- **`.`:** （一个点）可表示任何字符。

- `%a`：表示任何字母。
- `%c`：表示任何控制字符。
- `%d`：表示任何数字。
- `%g`：表示任何除空白符外的可打印字符。
- `%l`：表示所有小写字母。
- `%p`：表示所有标点符号。
- `%s`：表示所有空白字符。
- `%u`：表示所有大写字母。
- `%w`：表示所有字母及数字。
- `%x`：表示所有 16 进制数字符号。
- `%_x_`：（这里的 *x* 是任意非字母或数字的字符） 表示字符 *x*。 这是对魔法字符转义的标准方法。 所有非字母或数字的字符 （包括所有标点，也包括非魔法字符） 都可以用前置一个 '`%`' 放在模式串中表示自身。
- `[_set_]`：表示 *set* 中所有字符的联合。 可以以 '`-`' 连接，升序书写范围两端的字符来表示一个范围的字符集。 上面提到的 `%` *x* 形式也可以在 *set* 中使用 表示其中的一个元素。 其它出现在 *set* 中的字符则代表它们自己。 例如，`[%w_]` （或 `[_%w]`） 表示所有的字母数字加下划线）， `[0-7]` 表示 8 进制数字， `[0-7%l%-]` 表示 8 进制数字加小写字母与 '`-`' 字符。

  交叉使用类和范围的行为未定义。 因此，像 `[%a-z]` 或 `[a-%%]` 这样的模式串没有意义。

- `[^_set_]`：表示 *set* 的补集， 其中 *set* 如上面的解释。

所有单个字母表示的类别（`%a`，`%c`，等）， 若将其字母改为大写，均表示对应的补集。 例如， `%S` 表示所有非空格的字符。

如何定义字母、空格、或是其他字符组取决于当前的区域设置。 特别注意：`[a-z]` 未必等价于 `%l` 。

模式条目：

模式条目 可以是

- 单个字符类匹配该类别中任意单个字符；
- 单个字符类跟一个 '`*`'， 将匹配零或多个该类的字符。 这个条目总是匹配尽可能长的串；
- 单个字符类跟一个 '`+`'， 将匹配一或更多个该类的字符。 这个条目总是匹配尽可能长的串；
- 单个字符类跟一个 '`-`'， 将匹配零或更多个该类的字符。 和 '`*`' 不同， 这个条目总是匹配尽可能短的串；
- 单个字符类跟一个 '`?`'， 将匹配零或一个该类的字符。 只要有可能，它会匹配一个；
- `%_n_`， 这里的 *n* 可以从 1 到 9； 这个条目匹配一个等于 *n* 号捕获物（后面有描述）的

子串。

- `%b_xy_`， 这里的 *x* 和 *y* 是两个明确的字符； 这个条目匹配以 *x* 开始 *y* 结束， 且其中 *x* 和 *y* 保持 平衡 的字符串。 意思是，如果从左到右读这个字符串，对每次读到一个 *x* 就 *+1* ， 读到一个 *y* 就 *-1*， 最终结束处的那个 *y* 是第一个记数到 0 的 *y*。 举个例子，条目 `%b()` 可以匹配到括号平衡的表达式。
- `%f[_set_]`， 指 边境模式； 这个条目会匹配到一个位于 *set* 内某个字符之前的一个空串， 且这个位置的前一个字符不属于 *set* 。 集合 *set* 的含义如前面所述。 匹配出的那个空串之开始和结束点的计算就看成该处有个字符 '`\0`' 一样。

模式：

模式 指一个模式条目的序列。 在模式最前面加上符号 '`^`' 将锚定从字符串的开始处做匹配。 在模式最后面加上符号 '`$`' 将使匹配过程锚定到字符串的结尾。 如果 '`^`' 和 '`$`' 出现在其它位置，它们均没有特殊含义，只表示自身。

捕获：

模式可以在内部用小括号括起一个子模式； 这些子模式被称为 捕获物。 当匹配成功时，由 捕获物 匹配到的字符串中的子串被保存起来用于未来的用途。 捕获物以它们左括号的次序来编号。 例如，对于模式 `"(a*(.)%w(%s*))"` ， 字符串中匹配到 `"a*(.)%w(%s*)"` 的部分保存在第一个捕获物中 （因此是编号 1 ）； 由 "`.`" 匹配到的字符是 2 号捕获物， 匹配到 "`%s*`" 的那部分是 3 号。

作为一个特例，空的捕获 `()` 将捕获到当前字符串的位置（它是一个数字）。 例如，如果将模式 `"()aa()"` 作用到字符串 `"flaaap"` 上，将产生两个捕获物： 3 和 5 。

6.4.2 – 打包和解包用到的格式串

用于 `string.pack` ， `string.packsize` ， `string.unpack` 的第一个参数。 它是一个描述了需要创建或读取的结构之布局。

格式串是由转换选项构成的序列。 这些转换选项列在后面：

- **`<`**：设为小端编码
- **`>`**：设为大端编码
- **`=`**：大小端遵循本地设置
- **`![_n_]`**：将最大对齐数设为 `n` （默认遵循本地对齐设置）
- **`b`**：一个有符号字节 (`char`)
- **`B`**：一个无符号字节 (`char`)
- **`h`**：一个有符号 `short` （本地大小）
- **`H`**：一个无符号 `short` （本地大小）
- **`l`**：一个有符号 `long` （本地大小）
- **`L`**：一个无符号 `long` （本地大小）

- **j**：一个 lua_Integer
- **J**：一个 lua_Unsigned
- **T**：一个 size_t （本地大小）
- **i[_n_]**：一个 n 字节长（默认为本地大小）的有符号 int
- **I[_n_]**：一个 n 字节长（默认为本地大小）的无符号 int
- **f**：一个 float （本地大小）
- **d**：一个 double （本地大小）
- **n**：一个 lua_Number
- **c_n_**： n 字节固定长度的字符串
- **z**：零结尾的字符串
- **s[_n_]**：长度加内容的字符串，其长度编码为一个 n 字节（默认是个 size_t ） 长的无符号整数。
- **x**：一个字节的填充
- **X_op_**：按选项 op 的方式对齐（忽略它的其它方面）的一个空条目
- **'':** （空格）忽略

（" [_n_] "表示一个可选的整数。） 除填充、空格、配置项（选项 " xX <=>! "）外，每个选项都关联一个参数（对于 string.pack ） 或结果（对于 string.unpack ）。

对于选项 " !_n_ ", " s_n_ ", " i_n_ ", " I_n_ ", n 可以是 1 到 16 间的整数。 所有的整数选项都将做溢出检查； string.pack 检查提供的值是否能用指定的字长表示； string.unpack 检查读出的值能否置入 Lua 整数中。

任何格式串都假设有一个 " !1= " 前缀， 即最大对齐为 1 （无对齐）且采用本地大小端设置。

对齐行为按如下规则工作： 对每个选项，格式化时都会填充一些字节直到数据从一个特定偏移处开始， 这个位置是该选项的大小和最大对齐数中较小的那个数的倍数； 这个较小值必须是 2 个整数次方。 选项 " c " 及 " z " 不做对齐处理； 选项 " s " 对对齐遵循其开头的整数。

string.pack 用零去填充 （ string.unpack 则忽略它）。

6.5 – UTF-8 支持

这个库提供了对 UTF-8 编码的基础支持。 所有的函数都放在表 utf8 中。 此库不提供除编码处理之外的任何 Unicode 支持。 所有需要了解字符含义的操作，比如字符分类， 都不在此范畴。

除非另有说明， 当一个函数需要一个字节位置的参数时， 都假定这个位置要么从字节序列的开始计算， 要么从字符串长度加一的位置算。 和字符串库一样， 负的索引从字符串末尾计起。

utf8.char (···)

接收零或多个整数， 将每个整数转换成对应的 UTF-8 字节序列，并返回这些序列连接到一起的字符串。

`utf8.charpattern`

用于精确匹配到一个 UTF-8 字节序列的模式（是一个字符串，并非函数）" `[\0-\x7F\xC2-\xF4][\x80-\xBF]*` " （参见 §6.4.1）。 它假定处理的对象是一个合法的 UTF-8 字符串。

`utf8.codes (s)`

返回一系列的值，可以让

```
for p, c in utf8.codes(s) do _body_ end
```

迭代出字符串 `s` 中所有的字符。 这里的 `p` 是位置（按字节数）而 `c` 是每个字符的编号。 如果处理到一个不合法的字节序列，将抛出一个错误。

`utf8.codepoint (s [, i [, j]])`

一整数形式返回 `s` 中 从位置 `i` 到 `j` 间（包括两端） 所有字符的编号。 默认的 `i` 为 1 ，默认的 `j` 为 `i`。 如果碰上不合法的字节序列，抛出一个错误。

`utf8.len (s [, i [, j]])`

返回字符串 `s` 中 从位置 `i` 到 `j` 间 （包括两端） UTF-8 字符的个数。 默认的 `i` 为 1 ，默认的 `j` 为 -1 。 如果它找到任何不合法的字节序列， 返回假值加上第一个不合法字节的位置。

`utf8.offset (s, n [, i])`

返回编码在 `s` 中的第 `n` 个字符的开始位置（按字节数） （从位置 `i` 处开始统计）。 负 `n` 则取在位置 `i` 前的字符。 当 `n` 是非负数时，默认的 `i` 是 1， 否则默认为 `#s + 1`。 因此， `utf8.offset(s, -n)` 取字符串的倒数第 `n` 个字符的位置。 如果指定的字符不在其中或在结束点之后，函数返回 **nil**。

作为特例，当 `n` 等于 0 时， 此函数返回含有 `s` 第 `i` 字节的那个字符的开始位置。

这个函数假定 `s` 是一个合法的 UTF-8 字符串。

6.6 – 表处理

这个库提供了表处理的通用函数。 所有函数都放在表 `table` 中。

记住，无论何时，若一个操作需要取表的长度， 这张表必须是一个真序列，或是拥有 `__len` 元方法 （参见 §3.4.7 ）。 所有的函数都忽略传入参数的那张表中的非数字键。

`table.concat (list [, sep [, i [, j]]])`

提供一个列表，其所有元素都是字符串或数字，返回字符串 `list[i]..sep..list[i+1] ··· sep..list[j]`。 `sep` 的默认值是空串， `i` 的默认值是 1 ， `j` 的默认值是 `#list` 。 如果 `i` 比 `j` 大，返回空串。

`table.insert (list, [pos,] value)`

在 `list` 的位置 `pos` 处插入元素 `value` ， 并后移元素 `list[pos], list[pos+1], ···, list[#list]` 。 `pos` 的默认值为 `#list+1` ， 因此调用 `table.insert(t,x)` 会将 `x` 插在列表 `t` 的末尾。

`table.move (a1, f, e, t [,a2])`

将元素从表 `a1` 移到表 `a2` 。 这个函数做了次等价于后面这个多重赋值的等价操作： `a2[t],··· = a1[f],···,a1[e]` 。 `a2` 的默认值为 `a1` 。 目标区间可以和源区间重叠。 索引 `f` 必须是正数。

`table.pack (···)`

返回用所有参数以键 1,2, 等填充的新表， 并将 " `n` " 这个域设为参数的总数。 注意这张返回的表不一定是一个序列。

`table.remove (list [, pos])`

移除 `list` 中 `pos` 位置上的元素，并返回这个被移除的值。 当 `pos` 是在 1 到 `#list` 之间的整数时， 它向前移动元素 `list[pos+1], list[pos+2], ···, list[#list]` 并删除元素 `list[#list]` ； 索引 `pos` 可以是 `#list + 1` ，或在 `#list` 为 0 时可以是 0 ； 在这些情况下，函数删除元素 `list[pos]`。

`pos` 默认为 `#list`， 因此调用 `table.remove(l)` 将移除表 `l` 的最后一个元素。

`table.sort (list [, comp])`

在表内从 `list[1]` 到 `list[#list]` 原地 对其间元素按指定次序排序。 如果提供了 `comp` ， 它必须是一个可以接收两个列表内元素为参数的函数。 当第一个元素需要排在第二个元素之前时，返回真 （因此 `not comp(list[i+1],list[i])` 在排序结束后将为真）。 如果没有提供 `comp`， 将使用标准 Lua 操作 `<` 作为替代品。

排序算法并不稳定； 即当两个元素次序相等时，它们在排序后的相对位置可能会改变。

`table.unpack (list [, i [, j]])`

返回列表中的元素。 这个函数等价于

```
return list[i], list[i+1], ···, list[j]
```

`i` 默认为 1 ，`j` 默认为 `#list` 。

6.7 – 数学函数

这个库提供了基本的数学函数。 所以函数都放在表 `math` 中。 注解有 "`integer/float`" 的函数会对整数参数返回整数结果， 对浮点（或混合）参数返回浮点结果。 圆整函数（`math.ceil`，`math.floor`，`math.modf`） 在结果在整数范围内时返回整数，否则返回浮点数。

`math.abs (x)`

返回 `x` 的绝对值。(integer/float)

`math.acos (x)`

返回 `x` 的反余弦值（用弧度表示）。

`math.asin (x)`

返回 `x` 的反正弦值（用弧度表示）。

`math.atan (y [, x])`

返回 `y/x` 的反正切值（用弧度表示）。 它会使用两个参数的符号来找到结果落在哪个象限中。 （即使 `x` 为零时，也可以正确的处理。）

默认的 `x` 是 1 ， 因此调用 `math.atan(y)` 将返回 `y` 的反正切值。

`math.ceil (x)`

返回不小于 `x` 的最小整数值。

`math.cos (x)`

返回 `x` 的余弦（假定参数是弧度）。

`math.deg (x)`

将角 `x` 从弧度转换为角度。

`math.exp (x)`

返回 *e^x* 的值 （ `e` 为自然对数的底）。

`math.floor (x)`

返回不大于 `x` 的最大整数值。

`math.fmod (x, y)`

返回 `x` 除以 `y`，将商向零圆整后的余数。 (integer/float)

`math.huge`

浮点数 `HUGE_VAL`， 这个数比任何数字值都大。

`math.log (x [, base])`

返回以指定底的 `x` 的对数。 默认的 `base` 是 *e* （因此此函数返回 `x` 的自然对数）。

`math.max (x, ···)`

返回参数中最大的值， 大小由 Lua 操作 `<` 决定。 (integer/float)

`math.maxinteger`

最大值的整数。

`math.min (x, ···)`

返回参数中最小的值， 大小由 Lua 操作 `<` 决定。 (integer/float)

`math.mininteger`

最小值的整数。

`math.modf (x)`

返回 `x` 的整数部分和小数部分。 第二个结果一定是浮点数。

`math.pi`

π 的值。

`math.rad (x)`

将角 `x` 从角度转换为弧度。

`math.random ([m [, n]])`

当不带参数调用时， 返回一个 *[0,1)* 区间内一致分布的浮点伪随机数。 当以两个整数 `m` 与 `n` 调用时， `math.random` 返回一个 *[m, n]* 区间 内一致分布的整数伪随机数。 （值 *m-n* 不能是负数，且必须在 Lua 整数的表示范围内。） 调用 `math.random(n)` 等价于 `math.random(1,n)`。

这个函数是对 C 提供的位随机数函数的封装。 对其统计属性不作担保。

`math.randomseed (x)`

把 `x` 设为伪随机数发生器的“种子”： 相同的种子产生相同的随机数列。

`math.sin (x)`

返回 `x` 的正弦值（假定参数是弧度）。

`math.sqrt (x)`

返回 `x` 的平方根。 （你也可以使用乘方 `x^0.5` 来计算这个值。）

`math.tan (x)`

返回 `x` 的正切值（假定参数是弧度）。

`math.tointeger (x)`

如果 `x` 可以转换为一个整数， 返回该整数。 否则返回 **nil**。

`math.type (x)`

如果 `x` 是整数，返回 "`integer`"， 如果它是浮点数，返回 "`float`"， 如果 `x` 不是数字，返回 **nil**。

`math.ult (m, n)`

如果整数 `m` 和 `n` 以无符号整数形式比较， `m` 在 `n` 之下，返回布尔真否则返回假。

6.8 – 输入输出库

I/O 库提供了两套不同风格的文件处理接口。 第一种风格使用隐式的文件句柄； 它提供设置默认输入文件及默认输出文件的操作， 所有的输入输出操作都针对这些默认文件。 第二种风格使用显式的文件句柄。

当使用隐式文件句柄时， 所有的操作都由表 `io` 提供。 若使用显式文件句柄， `io.open` 会返回一个文件句柄，且所有的操作都由该文件句柄的方法来提供。

表 `io` 中也提供了三个 和 C 中含义相同的预定义文件句柄： `io.stdin` ， `io.stdout` ， 以及 `io.stderr` 。 I/O 库永远不会关闭这些文件。

除非另有说明， I/O 函数在出错时都返回 **nil** （第二个返回值为错误消息，第三个返回值为系统相关的错误码）。 成功时返回与 **nil** 不同的值。 在非 POSIX 系统上， 根据错误码取出错误消息的过程可能并非线程安全的， 因为这使用了 C 的全局变量 `errno` 。

`io.close ([file])`

等价于 `file:close()` 。 不给出 `file` 时将关闭默认输出文件。

`io.flush ()`

等价于 `io.output():flush()` 。

`io.input ([file])`

用文件名调用它时， （以文本模式）来打开该名字的文件， 并将文件句柄设为默认输入文件。 如果用文件句柄去调用它， 就简单的将该句柄设为默认输入文件。 如果调用时不传参数， 它返回当前的默认输入文件。

在出错的情况下，函数抛出错误而不是返回错误码。

`io.lines ([filename ···])`

以读模式打开指定的文件名并返回一个迭代函数。 此迭代函数的工作方式和用一个已打开的文件去调用 `file:lines(···)` 得到的迭代器相同。 当迭代函数检测到文件结束， 它不返回值（让循环结束）并自动关闭文件。

调用 `io.lines()` （不传文件名） 等价于 `io.input():lines("*l")` ； 即， 它将按行迭代标准输入文件。 在此情况下，循环结束后它不会关闭文件。

在出错的情况下，函数抛出错误而不是返回错误码。

`io.open (filename [, mode])`

这个函数用字符串 `mode` 指定的模式打开一个文件。 返回新的文件句柄。 当出错时，返回 **nil** 加错误消息。

`mode` 字符串可以是下列任意值：

- "`r`": 读模式（默认）；
- "`w`": 写模式；
- "`a`": 追加模式；
- "`r+`": 更新模式，所有之前的数据都保留；
- "`w+`": 更新模式，所有之前的数据都删除；
- "`a+`": 追加更新模式，所有之前的数据都保留，只允许在文件尾部做写入。

`mode` 字符串可以在最后加一个 '`b`'， 这会在某些系统上以二进制方式打开文件。

`io.output ([file])`

类似于 `io.input`。 不过都针对默认输出文件操作。

`io.popen (prog [, mode])`

这个函数和系统有关，不是所有的平台都提供。

用一个分离进程开启程序 `prog`， 返回的文件句柄可用于从这个程序中读取数据 （如果 `mode` 为 `"r"`， 这是默认值） 或是向这个程序写入输入（当 `mode` 为 `"w"` 时）。

`io.read (···)`

等价于 `io.input():read(···)`。

`io.tmpfile ()`

返回一个临时文件的句柄。 这个文件以更新模式打开，在程序结束时会自动删除。

`io.type (obj)`

检查 `obj` 是否是合法的文件句柄。 如果 `obj` 它是一个打开的文件句柄，返回字符串 `"file"`。 如果 `obj` 是一个关闭的文件句柄，返回字符串 `"closed file"`。 如果 `obj` 不是文件句柄，返回 **nil** 。

`io.write (···)`

等价于 `io.output():write(···)`。

`file:close ()`

关闭 `file` 。 注意，文件在句柄被垃圾回收时会自动关闭， 但是多久以后发生，时间不可预期的。

当关闭用 `io.popen` 创建出来的文件句柄时， `file:close` 返回 `os.execute` 会返回的一样的值。

`file:flush ()`

将写入的数据保存到 `file` 中。

`file:lines (···)`

返回一个迭代器函数， 每次调用迭代器时，都从文件中按指定格式读数据。 如果没有指定格式，使用默认值 " `l` " 。 看一个例子

```
     for c in file:lines(1) do _body_ end
```

会从文件当前位置开始，中不断读出字符。 和 `io.lines` 不同， 这个函数在循环结束后不会关闭文件。

在出错的情况下，函数抛出错误而不是返回错误码。

`file:read (···)`

读文件 `file` ， 指定的格式决定了要读什么。 对于每种格式，函数返回读出的字符对应的字符串或数字。 若不能以该格式对应读出数据则返回 **nil**。 （对于最后这种情况， 函数不会读出后续的格式。） 当调用时不传格式，它会使用默认格式读下一行（见下面描述）。

提供的格式有

- **"** `n` **"**: 读取一个数字，根据 Lua 的转换文法，可能返回浮点数或整数。 （数字可以有前置或后置的空格，以及符号。） 只要能构成合法的数字，这个格式总是去读尽量长的串； 如果读出来的前缀无法构成合法的数字 （比如空串，" `0x` " 或 " `3.4e-` "）， 就中止函数运行，返回 **nil**。
- **"** `i` **"**: 读取一个整数，返回整数值。
- **"** `a` **"**: 从当前位置开始读取整个文件。 如果已在文件末尾，返回空串。
- **"** `l` **"**: 读取一行并忽略行结束标记。 当在文件末尾时，返回 **nil** 这是默认格式。
- **"** `L` **"**: 读取一行并保留行结束标记（如果有的话）， 当在文件末尾时，返回 **nil**。
- ***number***: 读取一个不超过这个数量字节数的字符串。 当在文件末尾时，返回 **nil**。 如果 `number` 为零， 它什么也不读，返回一个空串。 当在文件末尾时，返回 **nil**。

格式 " `l` " 和 " `L` " 只能用于文本文件。

`file:seek ([whence [, offset]])`

设置及获取基于文件开头处计算出的位置。 设置的位置由 `offset` 和 `whence` 字符串 `whence` 指定的基点决定。基点可以是：

- " **set** ": 基点为 0 （文件开头）；
- " **cur** ": 基点为当前位置了；
- " **end** ": 基点为文件尾；

当 `seek` 成功时，返回最终从文件开头计算起的文件的位置。 当 `seek` 失败时，返回 **nil** 加上一个错误描述字符串。

`whence` 的默认值是 `"cur"`， `offset` 默认为 0 。 因此，调用 `file:seek()` 可以返回文件当前位置，并不改变它； 调用 `file:seek("set")` 将位置设为文件开头（并返回 0）； 调用 `file:seek("end")` 将位置设到文件末尾，并返回文件大小。

`file:setvbuf (mode [, size])`

设置输出文件的缓冲模式。 有三种模式：

- " **no** ": 不缓冲；输出操作立刻生效。
- " **full** ": 完全缓冲；只有在缓存满或当你显式的对文件调用 `flush` （参见 `io.flush` ） 时才真正做输出操作。
- " **line** ": 行缓冲； 输出将到每次换行前， 对于某些特殊文件（例如终端设备）缓冲到任何输入前。

对于后两种情况， `size` 以字节数为单位 指定缓冲区大小。 默认会有一个恰当的大小。

`file:write (···)`

将参数的值逐个写入 `file` 。 参数必须是字符串或数字。

成功时，函数返回 `file` 。 否则返回 **nil** 加错误描述字符串。

6.9 – 操作系统库

这个库都通过表 `os` 实现。

`os.clock ()`

返回程序使用的按秒计 CPU 时间的近似值。

`os.date ([format [, time]])`

返回一个包含日期及时刻的字符串或表。 格式化方法取决于所给字符串 `format` 。

如果提供了 `time` 参数， 格式化这个时间 （这个值的含义参见 `os.time` 函数）。 否则， `date` 格式化当前时间。

如果 `format` 以 '`!`' 打头， 日期以协调世界时格式化。 在这个可选字符项之后， 如果 `format` 为字符串 "`*t`"， `date` 返回有后续域的表： `year` （四位数字）， `month` （1–12）， `day` （1–31）， `hour` （0–23）， `min` （0–59）， `sec` （0–61）， `wday` （星期几，星期天为 1 ）， `yday` （当年的第几天）， 以及 `isdst` （夏令时标记，一个布尔量）。 对于最后一个域，如果该信息不提供的话就不存在。

如果 `format` 并非 "`*t`"， `date` 以字符串形式返回， 格式化方法遵循 ISO C 函数 `strftime` 的规则。

如果不传参数调用， `date` 返回一个合理的日期时间串， 格式取决于宿主程序以及当前的区域设置 （即， `os.date()` 等价于 `os.date("%c")` ）。

在非 POSIX 系统上， 由于这个函数依赖 C 函数 `gmtime` 和 `localtime` ， 它可能并非线程安全的。

`os.difftime (t2, t1)`

返回以秒计算的时刻 `t1` 到 `t2` 的差值。 （这里的时刻是由 `os.time` 返回的值）。 在 POSIX，Windows，和其它一些系统中，这个值就等于 `t2` - `t1` 。

`os.execute ([command])`

这个函数等价于 ISO C 函数 `system` 。 它调用系统解释器执行 `command` 。 如果命令成功运行完毕，第一个返回值就是 **true**， 否则是 **nil** otherwise。 在第一个返回值之后，函数返回一个字符串加一个数字。如下：

- **"`exit`"**: 命令正常结束； 接下来的数字是命令的退出状态码。
- **"`signal`"**: 命令被信号打断； 接下来的数字是打断该命令的信号。

如果不带参数调用， `os.execute` 在系统解释器存在的时候返回真。

`os.exit ([code [, close]])`

调用 ISO C 函数 `exit` 终止宿主程序。 如果 `code` 为 **true**， 返回的状态码是 `EXIT_SUCCESS` ； 如果 `code` 为 **false**， 返回的状态码是 `EXIT_FAILURE` ； 如果 `code` 是一个数字， 返回的状态码就是这个数字。 `code` 的默认值为 **true**。

如果第二个可选参数 `close` 为真， 在退出前关闭 Lua 状态机。

`os.getenv (varname)`

返回进程环境变量 `varname` 的值， 如果该变量未定义，返回 **nil** 。

`os.remove (filename)`

删除指定名字的文件（在 POSIX 系统上可以是一个空目录） 如果函数失败，返回 **nil** 加一个错误描述串及出错码。

`os.rename (oldname, newname)`

将名字为 `oldname` 的文件或目录更名为 `newname` 。 如果函数失败，返回 **nil** 加一个错误描述串及出错码。

`os.setlocale (locale [, category])`

设置程序的当前区域。 `locale` 是一个区域设置的系统相关字符串； `category` 是一个描述有改变哪个分类的可选字符串： `"all"`， `"collate"`， `"ctype"`， `"monetary"`， `"numeric"`， 或 `"time"`； 默认的分类为 `"all"`。 此函数返回新区域的名字。 如果请求未被获准，返回 **nil** 。

当 `locale` 是一个空串， 当前区域被设置为一个在实现中定义好的本地区域。 当 `locale` 为字符串 "`C`"， 当前区域被设置为标准 C 区域。

当第一个参数为 **nil** 时， 此函数仅返回当前区域指定分类的名字。

由于这个函数依赖 C 函数 `setlocale`， 它可能并非线程安全的。

`os.time ([table])`

当不传参数时，返回当前时刻。 如果传入一张表，就返回由这张表表示的时刻。 这张表必须包含域 `year`， `month`， 及 `day`； 可以包含有 `hour` （默认为 12 ）， `min` （默认为 0）， `sec` （默认为 0）， 以及 `isdst` （默认为 **nil**）。 关于这些域的详细描述，参见 `os.date` 函数。

返回值是一个含义由你的系统决定的数字。 在 POSIX，Windows，和其它一些系统中， 这个数字统计了从指定时间（"epoch"）开始经历的秒数。 对于另外的系统，其含义未定义， 你只能把 `time` 的返回数字用于 `os.date` 和 `os.difftime` 的参数。

`os.tmpname ()`

返回一个可用于临时文件的文件名字符串。 这个文件在使用前必须显式打开，不再使用时需要显式删除。

在 POSIX 系统上， 这个函数会以此文件名创建一个文件以回避安全风险。 （别人可能未经允许在获取到这个文件名到创建该文件之间的时刻创建此文件。） 你依旧需要在使用它的时候先打开，并最后删除（即使你没使用到）。

只有有可能，你更应该使用 `io.tmpfile`， 因为该文件可以在程序结束时自动删除。

6.10 – 调试库

这个库提供了 Lua 程序调试接口（§4.9）的功能。 其中一些函数违反了 Lua 代码的基本假定（例如，不会从函数之外访问函数的局部变量； 用户数据的元表不会被 Lua 代码修改； Lua 程序不会崩溃）， 因此它们有可能危害到其它代码的安全性。 此外，库里的一些函数可能运行的很慢。

这个库里的所有函数都提供在表 `debug` 内。 所有操作线程的函数，可选的第一个参数都是针对的线程。 默认值永远是当前线程。

`debug.debug ()`

进入一个用户交互模式，运行用户输入的每个字符串。 使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。 输入一行仅包含 `cont` 的字符串将结束这个函数， 这样调用者就可以继续向下运行。

注意， `debug.debug` 输入的命令在文法上并没有内嵌到任何函数中， 因此不能直接去访问局部变量。

`debug.gethook ([thread])`

返回三个表示线程钩子设置的值： 当前钩子函数，当前钩子掩码，当前钩子计数 （ `debug.sethook` 设置的那些）。

`debug.getinfo ([thread,] f [, what])`

返回关于一个函数信息的表。 你可以直接提供该函数， 也可以用一个数字 `f` 表示该函数。 数字 `f` 表示运行在指定线程的调用栈对应层次上的函数： 0 层表示当前函数（ `getinfo` 自身）； 1 层表示调用 `getinfo` 的函数 （除非是尾调用，这种情况不计入栈）；等等。 如果 `f` 是一个比活动函数数量还大的数字， `getinfo` 返回 **nil**。

只有字符串 `what` 中有描述要填充哪些项， 返回的表可以包含 `lua_getinfo` 能返回的所有项。 `what` 默认是返回提供的除合法行号表外的所有信息。 对于选项 '`f`'， 会在可能的情况下，增加 `func` 域保存函数自身。 对于选项 '`L`'， 会在可能的情况下，增加 `activelines` 域保存合法行号表。

例如,表达式 `debug.getinfo(1,"n")` 返回带有当前函数名字信息的表（如果找的到名字的话）， 表达式 `debug.getinfo(print)` 返回关于 `print` 函数的 包含有所有能提供信息的表。

`debug.getlocal ([thread,] f, local)`

此函数返回在栈的 `f` 层处函数的索引为 `local` 的局部变量 的名字和值。 这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。

第一个形参或是定义的第一个局部变量的索引为 1， 然后遵循在代码中定义次序，以次类推。 其中只计算函数当前作用域的活动变量。 负索引指可变参数； -1 指第一个可变参数。 如果该索引处没有变量，函数返回 **nil**。 若指定的层次越界，抛出错误。 （你可以调用 `debug.getinfo` 来检查层次是否合法。）

以 '`(`' （开括号）打头的变量名表示没有名字的变量 （比如是循环控制用到的控制变量， 或是去除了调试信息的代码块）。

参数 `f` 也可以是一个函数。 这种情况下， `getlocal` 仅返回函数形参的名字。

`debug.getmetatable (value)`

返回给定 `value` 的元表。 若其没有元表则返回 **nil** 。

`debug.getregistry ()`

返回注册表（参见 §4.5）。

`debug.getupvalue (f, up)`

此函数返回函数 `f` 的第 `up` 个上值的名字和值。 如果该函数没有那个上值，返回 **nil** 。

以 '`(`' （开括号）打头的变量名表示没有名字的变量 （去除了调试信息的代码块）。

`debug.getuservalue (u)`

返回关联在 `u` 上的 Lua 值。 如果 `u` 并非用户数据，返回 **nil**。

`debug.sethook ([thread,] hook, mask [, count])`

将一个函数作为钩子函数设入。 字符串 `mask` 以及数字 `count` 决定了钩子将在何时调用。 掩码是由下列字符组合成的字符串，每个字符有其含义：

- '**`c`**': 每当 Lua 调用一个函数时，调用钩子；
- '**`r`**': 每当 Lua 从一个函数内返回时，调用钩子；
- '**`l`**': 每当 Lua 进入新的一行时，调用钩子。

此外， 传入一个不为零的 `count` ， 钩子将在每运行 `count` 条指令时调用。

如果不传入参数， `debug.sethook` 关闭钩子。

当钩子被调用时， 第一个参数是触发这次调用的事件： `"call"` （或 `"tail call"` ）， `"return"` ， `"line"` ， `"count"` 。 对于行事件， 钩子的第二个参数是新的行号。 在钩子内，你可以调用 `getinfo` ，指定第 2 层， 来获得正在运行的函数的详细信息 （0 层指 `getinfo` 函数， 1 层指钩子函数）。

`debug.setlocal ([thread,] level, local, value)`

这个函数将 `value` 赋给 栈上第 `level` 层函数的第 `local` 个局部变量。 如果没有那个变量，函数返回 **nil** 。 如果 `level` 越界，抛出一个错误。 （你可以调用 `debug.getinfo` 来检查层次是否合法。） 否则，它返回局部变量的名字。

关于变量索引和名字，参见 `debug.getlocal` 。

`debug.setmetatable (value, table)`

将 `value` 的元表设为 `table` （可以是 **nil**）。 返回 `value` 。

`debug.setupvalue (f, up, value)`

这个函数将 `value` 设为函数 `f` 的第 `up` 个上值。 如果函数没有那个上值，返回 **nil** 否则，返回该上值的名字。

`debug.setuservalue (udata, value)`

将 `value` 设为 `udata` 的关联值。 `udata` 必须是一个完全用户数据。

返回 `udata` 。

`debug.traceback ([thread,] [message [, level]])`

如果 `message` 有，且不是字符串或 **nil**， 函数不做任何处理直接返回 `message` 。 否则，它返回调用栈的栈回溯信息。 字符串可选项 `message` 被添加在栈回溯信息的开头。 数字可选项 `level` 指明从栈的哪一层开始回溯 （默认为 1 ， 即调用 `traceback` 的那里）。

`debug.upvalueid (f, n)`

返回指定函数第 `n` 个上值的唯一标识符（一个轻量用户数据）。

这个唯一标识符可以让程序检查两个不同的闭包是否共享了上值。 若 Lua 闭包之间共享的是同一个上值 （即指向一个外部局部变量）， 会返回相同的标识符。

`debug.upvaluejoin (f1, n1, f2, n2)`

让 Lua 闭包 `f1` 的第 `n1` 个上值 引用 Lua 闭包 `f2` 的第 `n2` 个上值。

7 – 独立版 Lua

虽然 Lua 被设计成一门扩展式语言，用于嵌入一个宿主程序。 但经常也会被当成独立语言使用。 独立版的 Lua 语言解释器随标准包发布，就叫 `lua` 。 独立版解释器保留了所有的标准库及调试库。 其命令行用法为：

```
lua [options] [script [args]]
```

选项有：

- `-e _stat_` : 执行一段字符串 *stat* ；
- `-l _mod_` : "请求模块" *mod* ；
- `-i` : 在运行完 脚本 后进入交互模式；
- `-v` : 打印版本信息；
- `-E` : 忽略环境变量；
- `--` : 中止对后面选项的处理；
- `-` : 把 `stdin` 当作一个文件运行，并中止对后面选项的处理。

在处理完选项后， `lua` 运行指定的 脚本。 如果不带参数调用， 在标准输入（ `stdin` ）是终端时， `lua` 的行为和 `lua -v -i` 相同。 否则相当于 `lua -` 。

如果调用时不带选项 `-E` ， 解释器会在运行任何参数前，检查环境变量 `LUA_INIT_5_3` （或在版本名未定义时，检查 `LUA_INIT` ）。 如果该变量内存格式为 `@_filename_` ， `lua` 执行该文件。 否则， `lua` 执行该字符串。

如果调用时有选项 `-E` ， 除了忽略 `LUA_INIT` 外， Lua 还忽略 `LUA_PATH` 与 `LUA_CPATH` 的值。 将 `package.path` 和 `package.cpath` 的值设为定义在 `luaconf.h` 中的默认路径。

除 `-i` 与 `-E` 外所有的选项都按次序处理。 例如，这样调用

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

将先把 `a` 设为 1，然后打印 `a` 的值， 最后运行文件 `script.lua` 并不带参数。 （这里的 `$` 是命令行提示。你的命令行提示可能不一样。）

在运行任何代码前， `lua` 会将所有命令行传入的参数放到一张全局表 `arg` 中。 脚本的名字放在索引 0 的地方， 脚本名后紧跟的第一个参数在索引 1 处，依次类推。 在脚本名前面的任何参数 （即解释器的名字以及各选项） 放在负索引处。 例如，调用

```
$ lua -la b.lua t1 t2
```

这张表是这样的：

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

如果调用中没提供脚本名， 解释器的名字就放在索引 0 处，后面接着其它参数。 例如，调用

```
$ lua -e "print(arg[1])"
```

将打印出 " `-e` " 。 如果提供了脚本名， 就以 `arg[1]` , ···, `arg[#arg]` 为参数调用脚本。 （和 Lua 所有的代码块一样， 脚本被编译成一个可变参数函数。）

在交互模式下， Lua 不断的显示提示符，并等待下一行输入。 一旦读到一行， 首先试着把这行解释为一个表达式。 如果成功解释，就打印表达式的值。 否则，将这行解释为语句。 如果你写了一行未完成的语句， 解释器会用一个不同的提示符来等待你写完。

当脚本中出现了未保护的错误， 解释器向标准错误流报告错误。 如果错误对象并非一个字符串，但是却有元方法 `__tostring` 的话， 解释器会调用这个元方法生成最终的消息。 否则，解释器将错误对象转换为一个字符串，并把栈回溯信息加在前面。

如果正常结束运行， 解释器会关闭主 Lua 状态机 （参见 `lua_close` ） 。 脚本可以通过调用 `os.exit` 来结束，以回避这个步骤。

为了让 Lua 可以用于 Unix 系统的脚本解释器。 独立版解释器会忽略代码块的以 `#` 打头的第一行。 因此，Lua 脚本可以通过 `chmod +x` 以及 `#!` 形式变成一个可执行文件。 类似这样

```
#!/usr/local/bin/lua
```

（当然， Lua 解释器的位置对于你的机器来说可能不一样。 如果 `lua` 在你的 `PATH` 中， 写成

```
#!/usr/bin/env lua
```

更为通用。）

8 – 与之前版本不兼容的地方

这里我们列出了把程序从 Lua 5.2 迁移到 Lua 5.3 会碰到的不兼容的地方。 你可以在编译 Lua 时定义一些恰当的选项（参见文件 `luaconf.h` ）， 来回避一些不兼容性。 然而，这些兼容选项以后会移除。

Lua 的版本更替总是会修改一些 C API 并涉及源代码的改变。 例如一些常量的数字值，用宏来实现一些函数。 因此，你不能假设在不同的 Lua 版本间可以做到二进制兼容。 当你使用新版时，一定要将使用了 Lua API 的客户程序重新编译。

同样，Lua 版本更替还会改变预编译代码块的内部呈现方式；在不同的 Lua 版本间，预编译代码块不兼容。

官方发布版的标准路径也可能随版本变化。

8.1 – 语言的变更

- Lua 5.2 到 Lua 5.3 最大的变化是引入了数字的整数子类型。 虽然这个变化不会影响“一般”计算， 但一些计算 （主要是涉及溢出的） 会得到不同的结果。

  你可以通过把数字都强制转换为浮点数来消除差异 （在 Lua 5.2 中，所有的数字都是浮点数）。 比如你可以将所有的常量都以 `.0` 结尾， 或是使用 `x = x + 0.0` 来转换一个变量。 （这条建议仅用于偶尔快速解决一些不兼容问题；这不是一条好的编程准则。 好好写程序的话，你应该在需要使用浮点数的地方用浮点数， 需要整数的地方用整数。）

- 把浮点数转为字符串的地方，现在都对等于整数的浮点数加了 `.0` 后缀。 （例如，浮点数 2.0 会被打印成 `2.0`， 而不是 `2`。） 如果你需要定制数字的格式，就必须显式的格式化它们。

  （准确说这个不是兼容性问题， 因为 Lua 并没有规定数字如何格式化成字符串， 但一些程序假定遵循某种特别的格式。）

- 分代垃圾收集器没有了。 （它是 Lua 5.2 中的一个试验性特性。）

8.2 – 库的变更

- `bit32` 库废弃了。 使用一个外部兼容库很容易， 不过最好直接用对应的位操作符来替换它。 （注意 `bit32` 只能针对 32 位整数运算， 而标准 Lua 中的位操作可以用于 64 位整数。）
- 表处理库现在在读写其中的元素时会考虑元方法。
- `ipairs` 这个迭代器也会考虑元方法，而 `__ipairs` 元方法被废弃了。

- `io.read` 的选项名不再用 ' `*` ' 打头。 但出于兼容性考虑，Lua 会继续忽略掉这个字符。
- 数学库中的这些函数废弃了： `atan2` ， `cosh` ， `sinh` ， `tanh` ， `pow` ， `frexp` ， 以及 `ldexp` 。 你可以用 `x^y` 替换 `math.pow(x,y)；你可以用` math.atan `替换` math.atan2 `，前者现在可以接收一或两个参数； 你可以用` x 2.0^exp `替换` *math.ldexp(x,exp)* `。 若用到其它操作，你可以写一个扩展库，或在 Lua 中实现它们。`
  ` `require` 在搜索 C 加载器时处理版本号的方式有所变化。 现在，版本号应该跟在模块名后（其它大多数工具都是这样干的）。 出于兼容性考虑，如果使用新格式找不到加载器的话，搜索器依然会尝试旧格式。 （Lua 5.2 已经是这样处理了，但是并没有写在文档里。）`

`

8.3 – API 的变更

- 延续函数现在接收原来用 `lua_getctx` 获取的参数， 所以 `lua_getctx` 就去掉了。 按需要改写你的代码。
- 函数 `lua_dump` 有了一个额外的参数 `strip` 。 如果想和之前的行为一致， 这个值传 0 。
- 用于传入传出无符号整数的函数 （ `lua_pushunsigned` ， `lua_tounsigned` ， `lua_tounsignedx` ， `luaL_checkunsigned` ， `luaL_optunsigned` ） 都废弃了。 直接从有符号版做类型转换。
- 处理输入非默认整数类型的宏 （ `luaL_checkint` ， `luaL_optint` ， `luaL_checklong` ， `luaL_optlong` ） 废弃掉了。 直接使用 `lua_Integer` 加一个类型转换就可以替代 （或是只要有可能，就在你的代码中使用 `lua_Integer` ）。

9 – Lua 的完整语法

这是一份采用扩展 BNF 描述的 Lua 完整语法。 在扩展 BNF 中， {A} 表示 0 或多个 A， [A] 表示一个可选的 A。 （操作符优先级，参见 §3.4.8； 对于最终符号，名字，数字，字符串字面量的解释，参见 §3.1。）

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  '**;**' |
     varlist '**=**' explist |
     functioncall |
     label |
     **break** |
     **goto** Name |
     **do** block **end** |
     **while** exp **do** block **end** |
     **repeat** block **until** exp |
     **if** exp **then** block {**elseif** exp **then** block} [**else** block] **end
     **for** Name '**=**' exp '**,**' exp ['**,**' exp] **do** block **end** |
     **for** namelist **in** explist **do** block **end** |
     **function** funcname funcbody |
     **local** **function** Name funcbody |
     **local** namelist ['**=**' explist]

retstat ::= **return** [explist] ['**;**']

label ::= '**::**' Name '**::**'

funcname ::= Name {'**.**' Name} ['**:**' Name]

varlist ::= var {'**,**' var}

var ::=  Name | prefixexp '**[**' exp '**]**' | prefixexp '**.**' Name

namelist ::= Name {'**,**' Name}

explist ::= exp {'**,**' exp}

exp ::=  **nil** | **false** | **true** | Numeral | LiteralString | '**...**' | funct
     prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | '**(**' exp '**)**'

functioncall ::=  prefixexp args | prefixexp '**:**' Name args

args ::=  '**(**' [explist] '**)**' | tableconstructor | LiteralString

functiondef ::= **function** funcbody

funcbody ::= '**(**' [parlist] '**)**' block **end**

parlist ::= namelist ['**,**' '**...**'] | '**...**'

tableconstructor ::= '**{**' [fieldlist] '**}**'

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= '**[**' exp '**]**' '**=**' exp | Name '**=**' exp | exp

fieldsep ::= '**,**' | '**;**'

binop ::=  '**+**' | '**-**' | '*****' | '**/**' | '**//**' | '**^**' | '**%**' |
     '**&**' | '**~**' | '**|**' | '**>>**' | '**<<**' | '**..**' |
     '**<**' | '**<=**' | '**>**' | '**>=**' | '**==**' | '**~=**' |
     **and** | **or**

unop ::= '**-**' | **not** | '**#**' | '**~**'
```

Lua 5.3 术语中英对照表

译者在翻译时，使用了一些尚未统一的中文术语译词，统一罗列如下：

元表	**metatable**
元方法	metamethod
弱表	weak table
协程	coroutine
闭包	closure
注册表	registry
让出	yield
表	table
用户数据	userdata
延续点	continuation
宿主	host
一等公民	first-class
语法糖	syntactic sugar
序列	sequence
异构	heterogeneous
终结器	finalizer
暂时表	ephemeron table
符记	token
字面串	literal string
代码块	chunk
标签	label
实例化	instantiated
上值	upvalue

关于 Lua

Lua 是一门强大、快速、轻量的嵌入式脚本语言。它由巴西里约热内卢 Pontifical Catholic 大学的 PUC-Rio 团队 开发。 Lua 是一个 自由软件， 广泛应用于世界上无数产品和项目。

Lua 的 官方网站 上提供了关于 Lua 的完整信息， 包括 综合概要 和最新的 文档， 需要注意的是 参考手册 可能和 这里的版本 有所不同。

安装 Lua

Lua 以 源代码 的形式发布，使用之前，你需要构建它。 构建 Lua 非常简单，因为 Lua 是用纯粹的 ANSI C 实现的，在所有具备 ANSI C 编译器的平台都可以直接编译。 同时，Lua 也可以直接以 C++ 形式编译。 下面介绍了类 Unix 平台上的构建流程，另有 其它系统构建介绍 与 配置选项 以作参考。

如果你没有时间或兴趣自己编译 Lua， 可以从 LuaBinaries 获取编译后的二进制文件 或者从 LuaDist 这里获取 Lua 的多平台发布版（自带电池）。

构建 Lua

在大多数类 Unix 平台上，输入 " `make` " 加上合适的平台名即可。步骤如下：

1. 打开一个控制台窗口，切换到 `lua-5.3.0` 目录。 目录下的 `Makefile` 文件内包含了构建与安装流程。
2. 运行 " `make` " 并查看你的平台是否列在其中。 当前支持的平台有：

   aix bsd c89 freebsd generic linux macosx mingw posix solaris

   如果你的平台在其中，运行 " `make xxx` " 即可，xxx 代表你的平台名。

   如果你的平台不在其中，先尝试最相近的平台，再按 posix generic c89 顺序依次尝试。

3. 编译过程很短，最终在 `src` 目录下生成三个文件： lua (解释器), luac (编译器)和 liblua.a (静态库) 。

4. 构建完成后，可以运行 " `make test` " 来检查是否成功。 它会运行解释器并打印版本号。

如果你是 Linux 系统并出现了编译错误，请确认你是否安装了 `readline` （也可能叫 `libreadline-dev` 或者 `readline-devel` ） 开发包。 之后，如果还有链接错误，尝试 " `make linux MYLIBS=-ltermcap` " 。

安装 Lua

一旦你构建完毕，可能希望把 Lua 安装到系统默认位置， 那么执行 " `make install` " 即可。 系统默认位置以及如何安装都定义在 `Makefile` 中。 这个过程可能需要有相关的权限。

运行 " `make xxx install` " 可以构建和安装一步到位，xxx 指你的平台名。

如果你想把 Lua 安装在本地，运行 " `make local` "。 它会创建一个 `install` 目录，内有 `bin` , `include` , `lib` , `man` , `share` , 子目录，并将下列文件安装在其中。 如果你想安装到本地其它目录， 运行 " `make install INSTALL_TOP=xxx` "，xxx 指你选择的目录。 由于安装过程

中会切换到 `src` 以及 `doc` 目录进行， 所以当 `INSTALL_TOP` 不是绝对路径时务必小心。

```
bin:

lua luac

include:

lauxlib.h lua.h lua.hpp luaconf.h lualib.h

lib:

liblua.a

man/man1:

lua.1 luac.1
```

这些是开发时需要的目录。 如果你仅仅想运行一些 Lua 程序， 那么只需要 `bin` 和 `man` 下的文件。 `include` 和 `lib` 下的文件用于将 Lua 嵌入 C 或 C++ 程序。

定制

有三类定制，可以通过编辑文件完成：

- 怎样安装 Lua 以及安装到哪里 — 编辑 `Makefile` 。
- 怎样构建 Lua — 编辑 `src/Makefile` 。
- Lua 特性 — 编辑 `src/luaconf.h` 。

其实你不必编辑 Makefile 文件，make 的时候在命令行指定相关变量即可。 当然，编辑保存 Makefile 可以给定制留个记录。

另一方面，如果你需要定制一些 Lua 特性，那就需要在构建安装 Lua 前 编辑 `src/luaconf.h` 。 编辑过的文件必须确保一致性，也就是只安装在一个地方， 让所有你编译出来的用到 Lua 的程序都使用这唯一的这一份。 专家可以通过编辑 Lua 源代码来定制更多的东西。

在其它系统上构建 Lua

如果你不使用常规的 Unix 工具，那么构建 Lua 的流程就取决于你使用的编译器。 你需要创建若干工程来构建库，解释器以及编译器等。请把下列源文件加入相关工程：

库:

- lapi.c
- lcode.c
- lctype.c
- ldebug.c
- ldo.c
- ldump.c

- lfunc.c
- lgc.c
- llex.c
- lmem.c
- lobject.c
- lopcodes.c
- lparser.c
- lstate.c
- lstring.c
- ltable.c
- ltm.c
- lundump.c
- lvm.c
- lzio.c
- lauxlib.c
- lbaselib.c
- lbitlib.c
- lcorolib.c
- ldblib.c
- liolib.c
- lmathlib.c
- loslib.c
- lstrlib.c
- ltablib.c
- lutf8lib.c
- loadlib.c
- linit.c

解释器:

library, lua.c

编译器:

library, luac.c

把 Lua 以一个库形式用于你的程序，你需要知道如何用你的编译器创建库和使用库。 比如，以动态加载的 C 库形式使用 Lua，你需要了解如何创建动态库并让 Lua API 函数 在动态库中可见 — 不要 将 Lua 库链入每个动态库。 在 Unix 下，我们建议把 Lua 库静态链入宿主程序，然后将符号导出用于动态链接； `src/Makefile` 就是这样处理 Lua 解释器的。 在 Windows 下，我们建议把 Lua 库编译成一个 DLL 。 无论怎样，编译器 luac 都应该静态链接。

正如上面所述，你可以在构建 Lua 前编辑 `src/luaconf.h` 以定制一些特性。

自 Lua 5.2 以来的变更

这里列出了 Lua 5.3 引入的主要变更。 参考手册 中列出了 不兼容的地方。

主要变化

- 整数 (默认 64 位)
- 32 位整数的官方支持
- 位操作符
- 基本的 utf-8 支持
- 值的打包及解包函数

这些是 Lua 5.3 引入的其它变更：

语言

- 用户数据可以是任意 Lua 值
- 整数除法
- 某些元方法有了更加灵活的规则

库

- `ipairs` 以及表处理库都会考虑元方法
- `string.dump` 多了裁减选项
- 表处理库考虑了元方法
- 新函数 `table.move`
- 新函数 `string.pack`
- 新函数 `string.unpack`
- 新函数 `string.packsize`

C API

- 简化了延续点 API
- `lua_gettable` 以及类似函数会返回结果的值类型
- `lua_dump` 增加了裁减选项
- 新函数： `lua_geti`
- 新函数： `lua_seti`
- 新函数： `lua_isyieldable`
- 新函数： `lua_numbertointeger`

- 新函数： `lua_rotate`
- 新函数： `lua_stringtonumber`

Lua 独立解释器

- 可以做计算器使用；不再需要前置 '='
- `arg` 表对所有代码都可用

License



Lua is free software distributed under the terms of the MIT license reproduced below; it may be used for any purpose, including commercial purposes, at absolutely no cost without having to ask us. The only requirement is that if you do use Lua, then you should give us credit by including the appropriate copyright notice somewhere in your product or its documentation. For details, see this.